CS 594: SCIENTIFIC COMPUTING FOR ENGINEERS

PERFORMANCE ANALYSIS TOOLS

Gabriel Marin
gmarin@eecs.utk.edu

Some slides borrowed from Dan Terpstra, John Mellor-Crummey

Rough Outline

1. **Part I**
   - Motivation
   - Introduction to Computer Architecture
   - Overview of Performance Analysis techniques
2. **Part II**
   - Introduction to Hardware Counter Events
   - **PAPI**: Access to hardware performance counters
3. **Part III**
   - **HPCToolkit**: Low overhead, full code profiling using hardware counters sampling

What is Performance?

- Getting results as quickly as possible?
- Getting *correct* results as quickly as possible?
- What about Budget?
- What about Development Time?
- What about Hardware Usage?
- What about Power Consumption?

Why Performance Analysis

- Large investments in HPC systems
  - Procurement costs: ~$40 million / year
  - Operational costs: ~$5 million / year
  - Electricity costs: 1 MW year ~$1 million

- ***Efficient usage*** is important because of expensive and limited resources
- ***Scalability*** is important to achieve next bigger simulation

- Embedded systems have strict power and memory constraints.

Simple Performance Equation

$$t = N * C / f$$

- N – number of executed instructions
- C – CPI = cycles per instruction
- f – processor frequency

$$P = 1 / t = f * I / N$$

- I – IPC = instructions per cycle

- Frequency scaling provided “easy” performance gains for many years
  - Power use increases with frequency cubed

Simple Performance Equation

$$P = f * I / N$$

- N – affected by implementation algorithm, compiler, machine instruction set (e.g. SIMD instructions)
- f – determined by architecture, is not going up anymore
- I – affected by code optimizations (manual or compiler) and by micro-architecture optimizations

- Current architectures can issue 6-8 micro-ops per cycle
  - Retire 3-4 instructions per cycle (Itanium can retire 6)
  - IPC > 1.5 is very good, ~1 is OK, many applications get IPC < 1

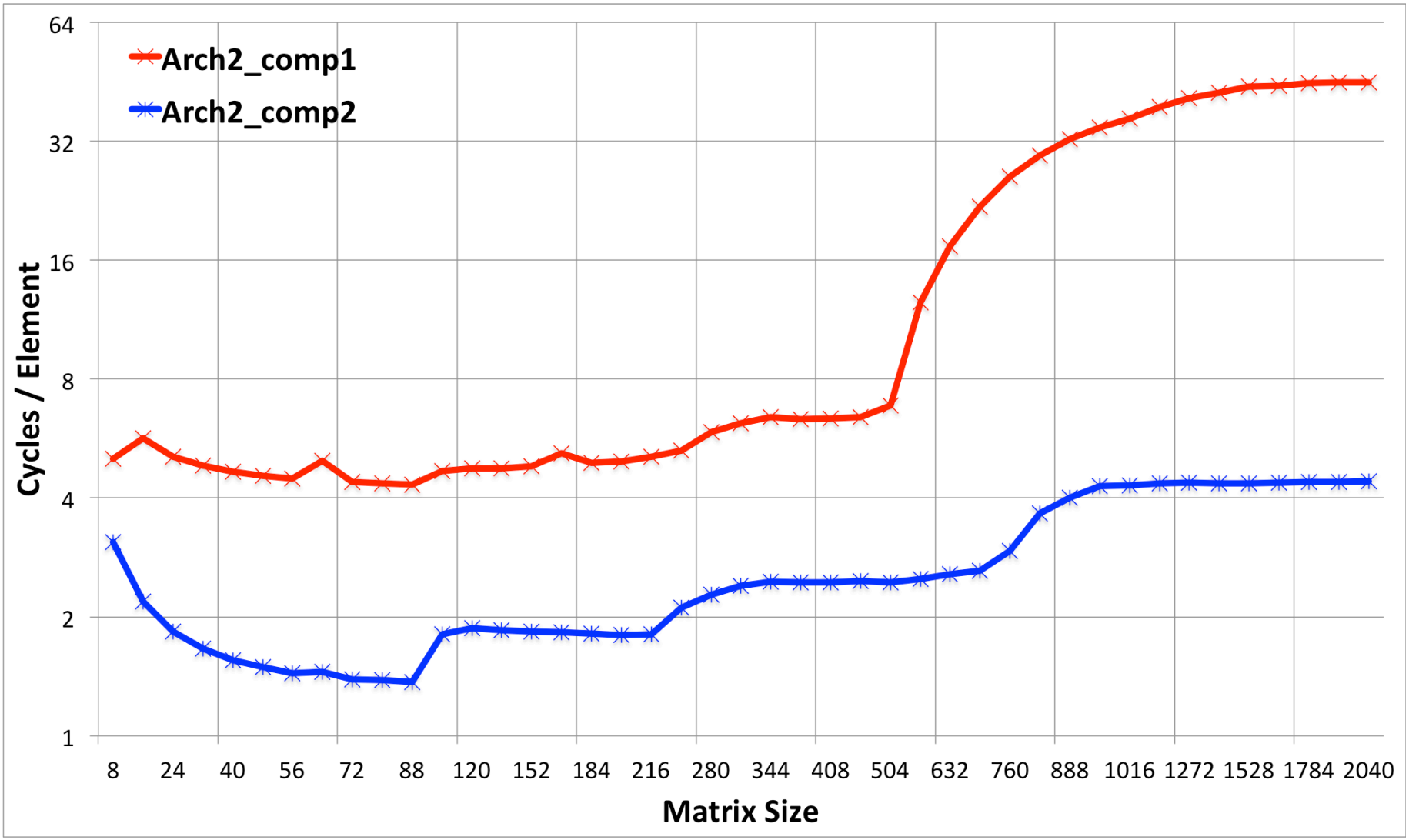
Factors Impacting Performance

- Algorithm – biggest impact
  - O(N*log(N)) performs much better than $O(N^2)$ for useful values of N
- Code implementation
  - Integer factor performance difference between efficient and inefficient implementations of the same algorithm
- Compiler and compiler flags
- Architecture

Example: Matrix Multiply

```
void compute(int reps) {
  int i, j, k, r;
  for (r=0 ; r<reps ; ++r)
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
           C(i,j) += A(i,k) * B(k,j);
}
```

Matrix Multiply: Different Compilers



Computer Architecture Review

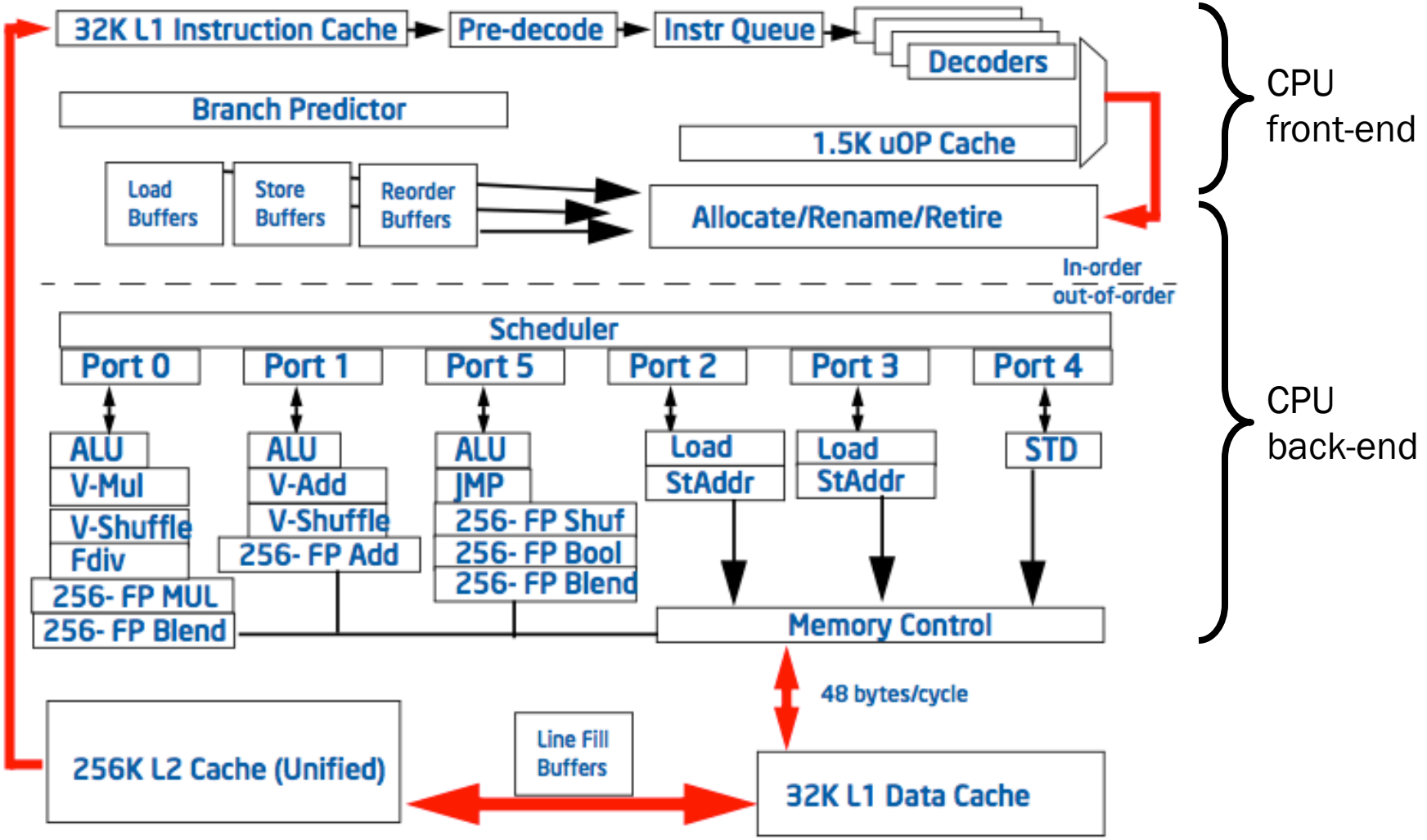
Knowledge of the micro-architecture is very important when trying to understand performance.

Most times the architecture is fixed, given to us.

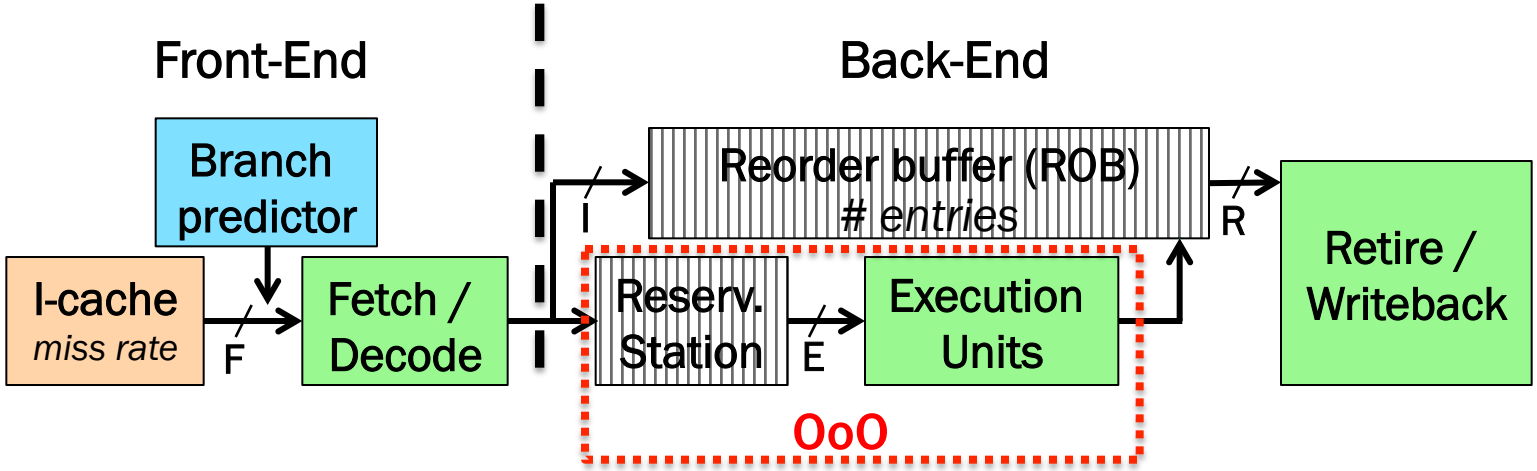
But we can tailor our code to the target architecture.

In very specialized cases we want to tailor an architecture to a specific application / workload

Computer Architecture Review

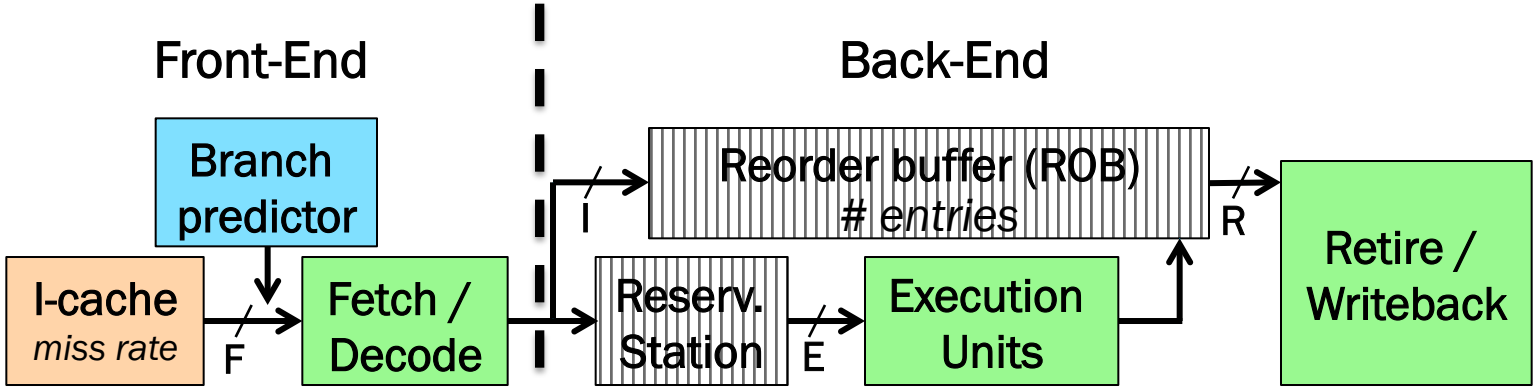


Processor Model



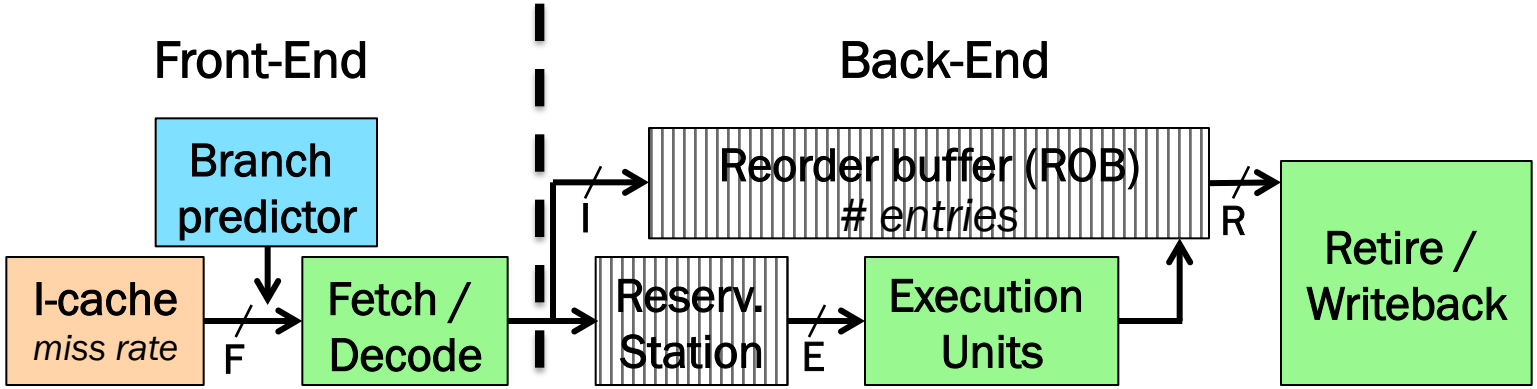
- Front-end: operates in-order
  - Instruction fetch/decode
  - Branch predictor - speculative instruction fetch and decode
    - ~ 1 in 5 instructions is a branch
- Back-end trumps front-end

Processor Front-End Stalls



- Possible front-end stall events
  - I-Cache or I-TLB miss
  - Branch misprediction
  - Full Reorder Buffer

FE Stalls: I-Cache or I-TLB Miss

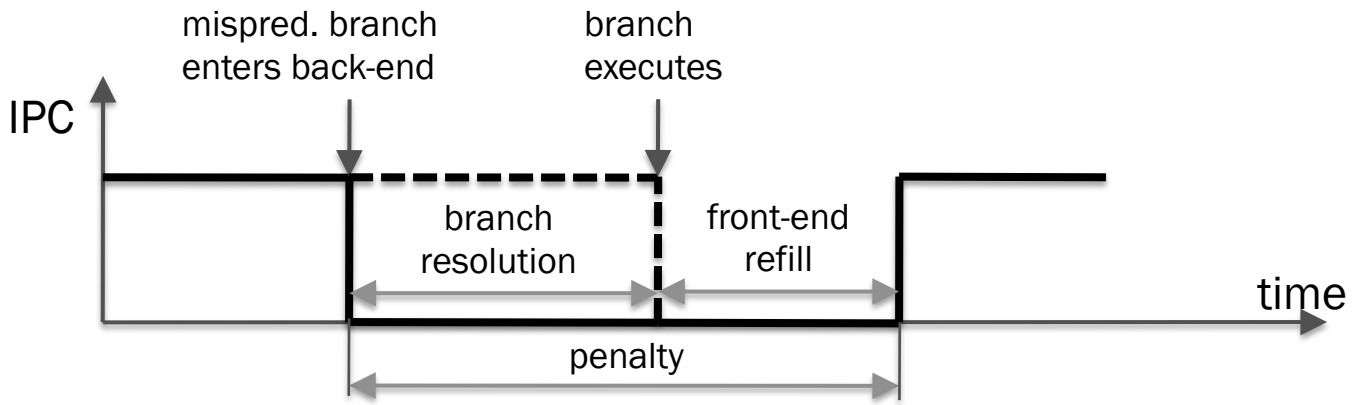


- Instruction fetch stops
- Instructions continue to be dispatched until buffers drains (hides penalty)
- Pipeline starts to refill once the miss event is resolved; refill time ~ drain time
- Penalty ~= miss event latency

FE Stalls: I-Cache or I-TLB Miss

- Possible causes
  - Execution spread over large regions of code with branchy unpredictable control flow
    - Not typical for HPC
  - Large loop footprint + small I-cache
    - Older Itanium2: 16KB I-cache, no hardware prefetcher
    - Space inefficient VLIW instruction set
    - Loop fusion / loop unrolling can create large loop footprints
- Possible solutions
  - Feedback directed compilation can change code layout
  - Limit loop unrolling or fusion

FE Stalls: Branch Misprediction



- Mispredicted branch is fetched
  - Instruction fetch continues along a wrong path
  - Pipeline filled with useful instructions at this point
- Mispredicted branch enters back-end
  - Back-end filled with useful instructions at this point
- Branch executes, misprediction detected, pipeline is flushed
- Instruction fetch starts on the correct path, front-end starts to refill
- Penalty ~= branch resolution time + front-end refill

Branch Misprediction Penalty

Architecture	Branch misprediction penalty
AMD K10 (Barcelona, Istanbul, Magny-Cours)	12
AMD Bulldozer	20
Pentium 4	20
Core 2 (Conroe, Penryn)	15
Nehalem	17
Sandy Bridge	14-17

- This is the minimum penalty, proportional to the processor pipeline depth.
  - Bulldozer has a deeper pipeline than K10 -> higher penalty
  - Sandy Bridge added a micro-ops cache, which can lower misprediction penalty compared to Nehalem.

Branch Misprediction Penalty

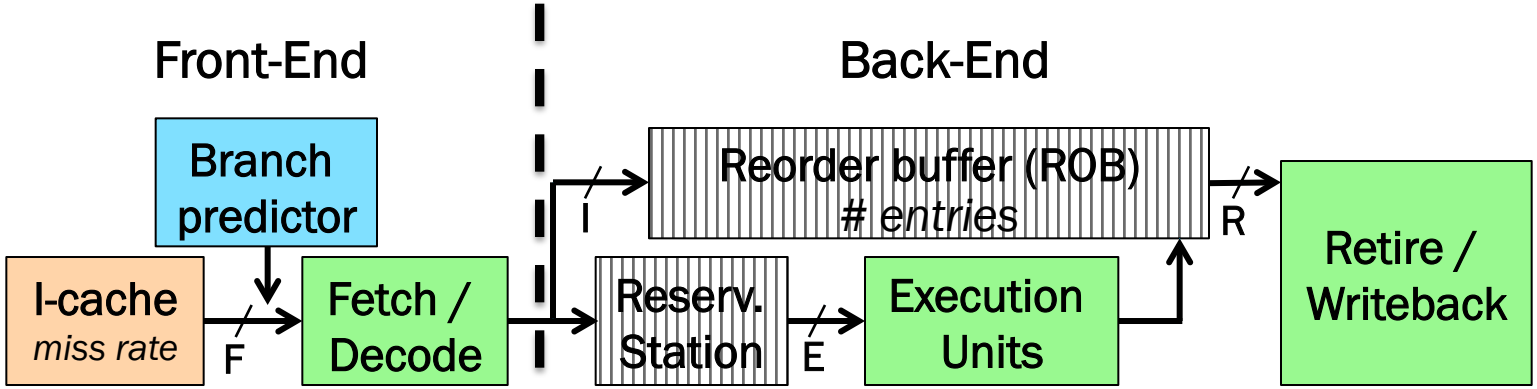
- Branch predictors have improved in time
  - Both Intel and AMD
- Modern branch predictors have very good accuracy on typical workloads, 95%+
- Is there room for improvement?
  - Does it matter if we go from 95% to 96%?

Branch Misprediction Penalty

- Branch predictors have improved in time
  - Both Intel and AMD
- Modern branch predictors have very good accuracy on typical workloads, 95%+
- Is there room for improvement?
  - Does it matter if we go from 95% to 96%?

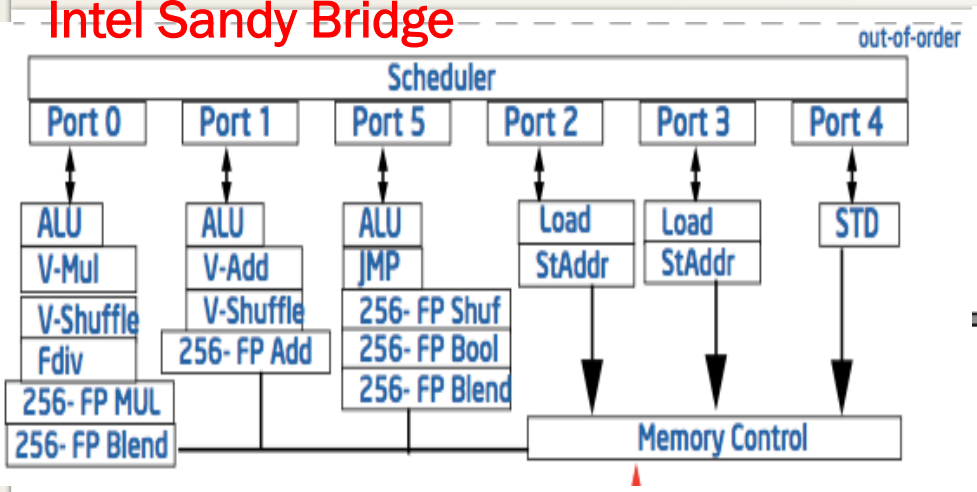
- Performance loss is proportional to branch misprediction rate
  - 5% to 4% misprediction rate is a 20% improvement
  - ~ 1 in 5 instructions is a branch in typical workloads
- Losses due to branch misprediction
  - Branch misprediction rate X pipeline depth

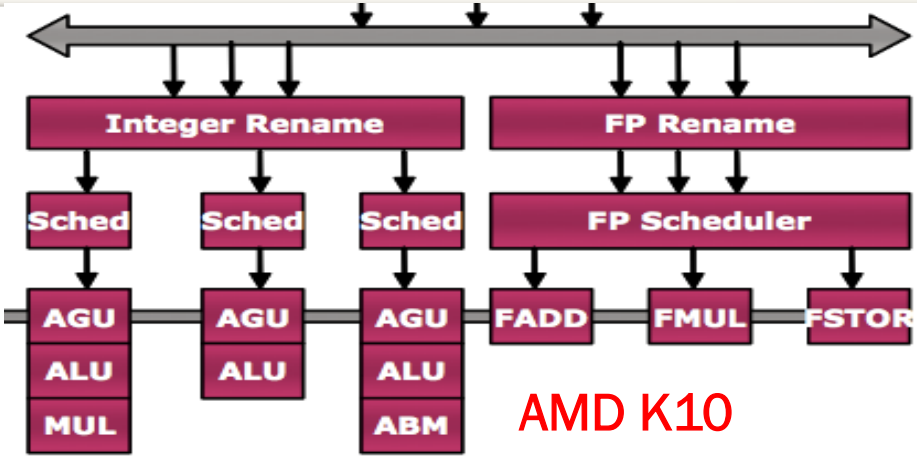
FE Stalls: Full ROB



- ROB maintains in-order state of not yet retired micro-ops
  - µops still in the issue buffer (have not executed)
  - µops in the back-end EU (executing)
  - µops that have completed before the ROB head micro-op
- On a long data access, other micro-ops continue to issue, but micro-ops dispatched after the stalled load cannot retire
- Dispatch continues until ROB fills up, then it stalls

Processor Back-End





- Execution units organized in stacks
  - Can issue one μop to each issue port each cycle
  - Can handle different instruction mixes
- Register files (not shown)
- Bypass network to forward results between stacks
- Back-end inefficiencies trump front-end

How To Define Peak Performance?

- Peak retirement rate (IPC)
  - From the architecture point of view
- Peak issue of “useful” instructions
  - HPC cares about FLOPS, mainly Adds and Multiplies
    - Peak FLOPS rate, everything else is overhead
  - You need many data movement instructions (Loads, Reg Copy, Data Shuffling, Data Conversion) + address arithmetic and Branches to perform useful work
    - Cannot get close to peak for most workloads, dense linear algebra is an exception
- What about SIMD instructions?
  - Peak issue of SIMD “useful” instructions

Back-End Inefficiencies

- Instruction dependencies limit available ILP
  - Machine units sit mostly idle
- Mismatch between application instruction mix and available machine resources
  - Contention on a particular execution unit or issue port
  - One unit heavily used while many units sit idle
- Too many overhead instructions: address arithmetic, spill / unspill code, branches, etc.
  - High IPC, but
  - “useful” operations are a fraction of all program operations

Back-End Inefficiencies

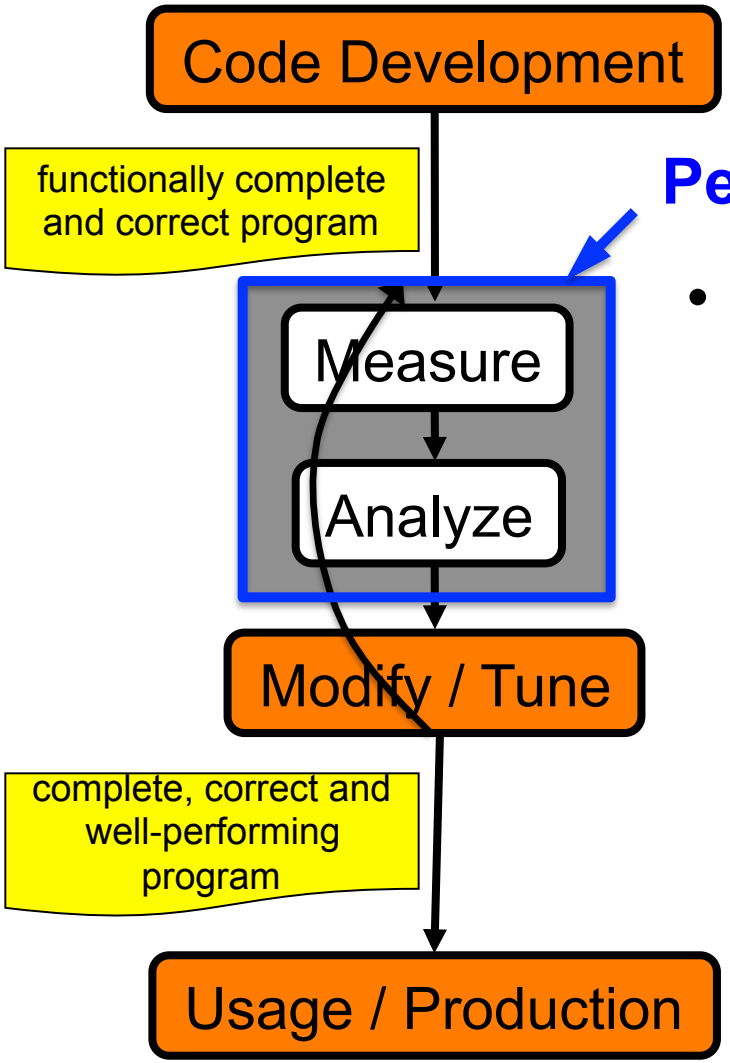
- Low mix of SIMD operations
  - Good IPC and ratio of “useful” instructions, but
  - Small fraction of peak performance
- Long data access
  - Memory access misses in D-Cache or D-TLB
  - Non-blocking caches, other instructions continue to issue
    - Multiple outstanding accesses to memory possible
  - Retirement stops on a long latency instruction
    - Eventually ROB fills up, dispatch stops

Long Data Accesses

- Typically the main source of performance losses
- Micro-architecture optimizations
  - Multiple levels of cache – exploit temporal and spatial reuse
    - Eliminate many accesses to memory
  - Hardware prefetchers – fetch data before it is needed
    - Hide memory latency
    - Work best with streaming memory access patterns
- Software optimizations
  - High level loop nest optimizations: tiling, fusion, loop interchange, loop splitting, data layout transformations
    - Increase temporal and/or spatial reuse
  - Software prefetching – uses instruction issue bandwidth

Introduction to Performance Analysis

Performance Optimization Cycle



Performance Analysis

- Understand
  - Where time is spent
  - What factors are limiting performance
  - Performance improvement potential

Performance Analysis Challenges

- Many variables affect performance
- Micro-architecture optimizations make analysis nondeterministic
- Performance measurement well supported by tools
- Analysis of results typically left to the user
- You must have a feeling of what can go wrong
  - Computer architecture insight helps
  - Everyone has a different style

Performance Analysis Techniques

- Performance measurement
- Performance modeling
- Simulation

- The line between different techniques can be blurry
  - Modeling can use measurement or simulation results as input

Performance Measurement

- Profiling vs. tracing
- Instrumentation vs. sampling

- Advantages
  - Performance of actual code on a real system
  - Reveals hotspots
- Disadvantages
  - Observes performance effects
    - Performance insight (diagnosis) not always apparent

Profiling vs. Tracing

Profiling

- Records aggregate performance metrics
  - No timeline dimension, or ordering of events
- Number of times a routine was invoked
- Time spent or cache misses incurred in a loop / routine

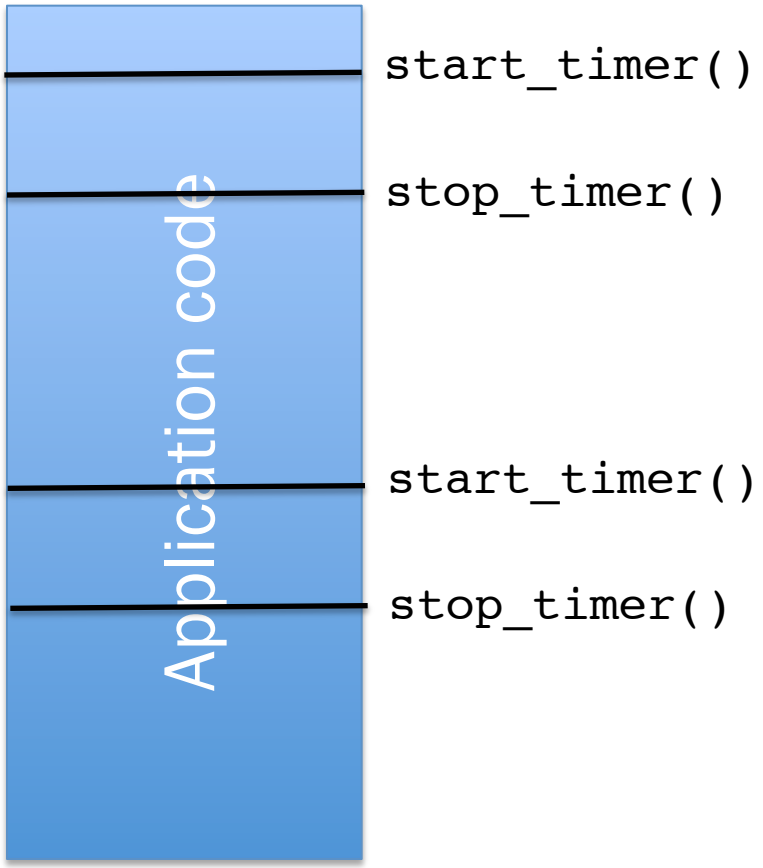
Tracing

- When and where events took place along a timeline
- Time-stamped events (points of interest)
- Shows when/where messages sent/received
- Event Trace: collection of all events of a process/program sorted by time

Profiling

- Recording of summary information during execution
  - inclusive, exclusive time, # calls, hardware counter statistics, …
- Reflects performance behavior of
  - program entities: functions, loops, basic blocks
  - user-defined “semantic” entities
- Very good for low-cost performance assessment
- Helps to expose hotspots
- Implemented through either
  - **instrumentation**: direct insertion of measurement code
  - **sampling**: periodic OS interrupts

Instrumentation



- Add calipers around code areas of interest
- Tradeoff
  - Instrumentation granularity vs. measurement overhead
  - Limits how much we can measure

Sampling

- OS interrupts at fixed intervals
  - Record program counter when interrupt received
  - Probability of recording a sample goes up with time spent in a given code region
- Collect distribution of interrupt samples
- Arbitrarily low overhead
- Full code coverage
- Some attribution errors

Tracing

- Record information about significant points (events) during program execution
- Save information in event record
  - Timestamp
  - CPU identifier, thread identifier
  - Event type and event-specific information
- Useful to expose interactions between parallel processes or threads

Tracing Disadvantages

- Traces can become very large
- Instrumentation and tracing add overhead
- Handle clock synchronization

Performance Analysis Tools

Raj Jain (1991)

> "Contrary to common belief, performance evaluation is an art. ... Like artist, each analyst has a unique style. Given the sample problem, two analysts may choose different performance metrics and evaluation methodologies."

… but even they need tools!

Outline

1. **Part I**
   - Motivation
   - Introduction to Computer Architecture
   - Overview of Performance Analysis techniques
2. **Part II**
   - Introduction to Hardware Counter Events
   - **PAPI**: Access to hardware performance counters
3. **Part III**
   - **HPCToolkit**: Low overhead, full code profiling using hardware counters sampling

What's PAPI?

- Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.
- Countable events are defined in two ways:
  - platform-neutral Preset Events
  - Platform-dependent Native Events
- Presets can be derived from multiple Native Events
- All events are referenced by name and collected in EventSets for sampling
- Events can be multiplexed if counters are limited
- Statistical sampling implemented by:
  - Hardware overflow if supported by the platform
  - Software overflow with timer driven sampling

Where's PAPI

- PAPI runs on most modern processors and Operating Systems of interest to HPC:
  - IBM
    - POWER series / AIX
    - POWER series, PowerPC / Linux
    - Blue Gene/L/P/Q
  - Intel / Linux
    - RAPL
    - MIC (Xeon PHI)
  - AMD / Linux
  - Cray / CrayPAT
  - ARM
  - NVIDIA / CUDA

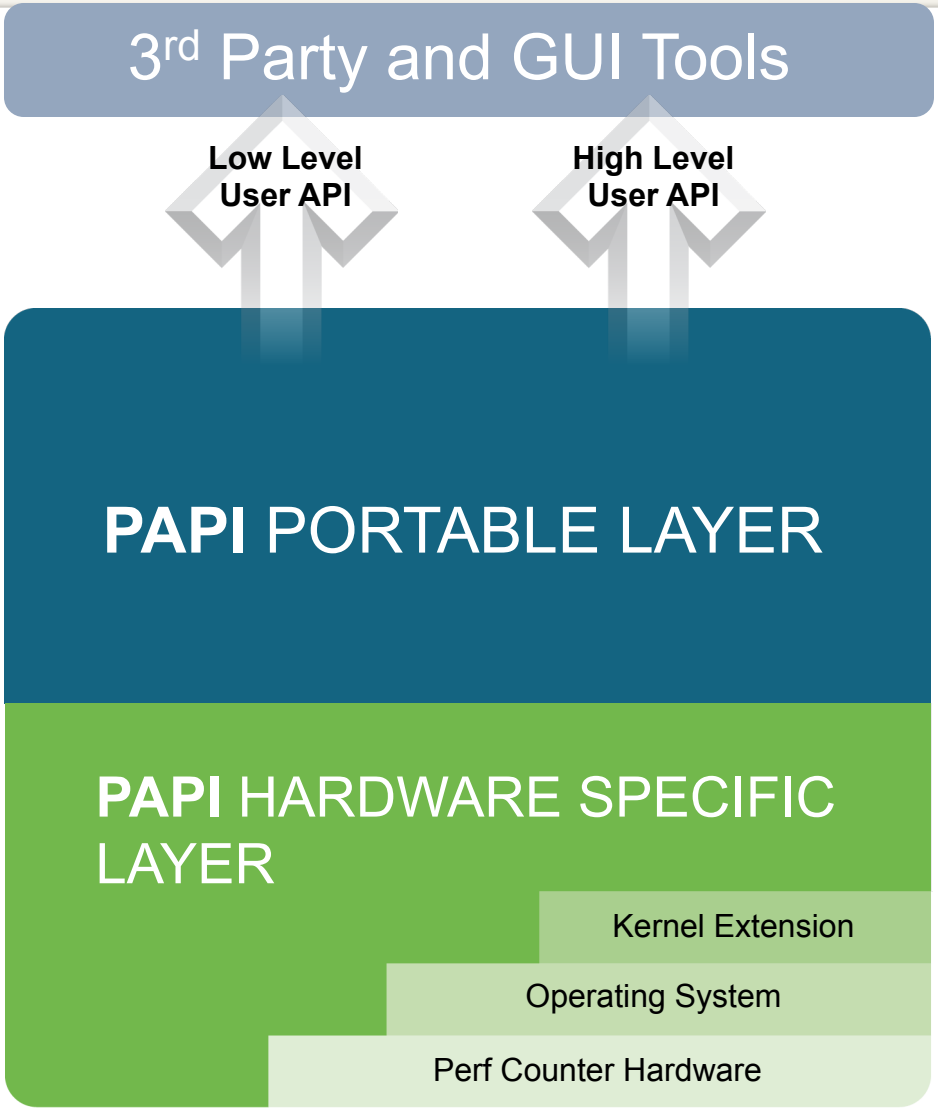
Some tools that use PAPI

- TAU (U Oregon)
  - http://www.cs.uoregon.edu/research/tau/
- PerfSuite (NCSA)
  - http://perfsuite.ncsa.uiuc.edu/
- Scalasca (UTK, FZ Juelich)
  - http://www.fz-juelich.de/jsc/scalasca/
- Vampir (TUDresden)
  - http://www.vampir.eu/
- HPCToolkit (Rice Univ.)
  - http://hpctoolkit.org/
- Open|Speedshop (SGI)
  - http://oss.sgi.com/projects/openspeedshop/

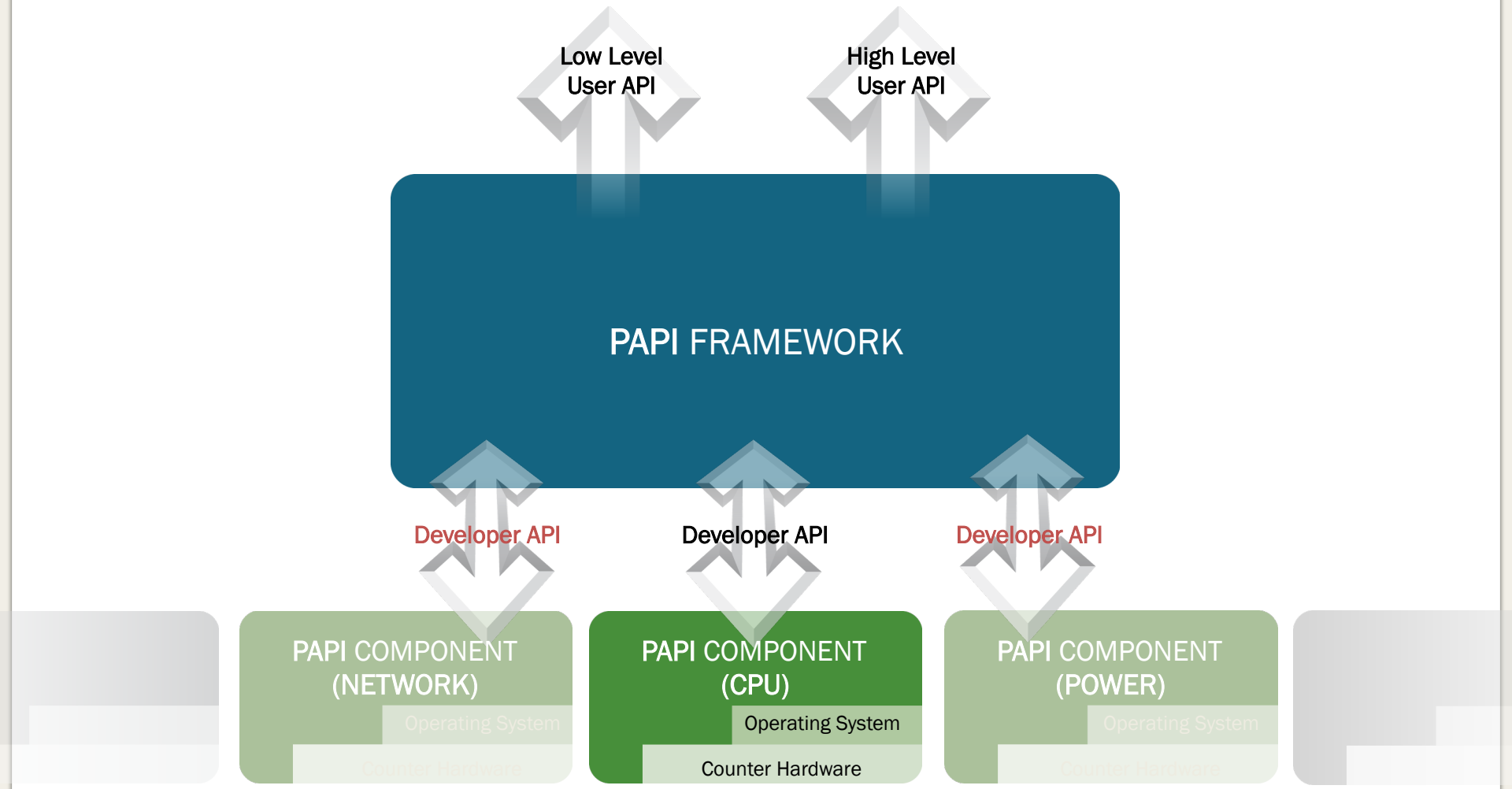
PAPI Counter Interfaces

PAPI provides 3 interfaces to the underlying counter hardware:

1. A Low Level API manages hardware events in user defined groups called *EventSets*, and provides access to advanced features.
2. A High Level API provides the ability to start, stop and read the counters for a specified list of events.
3. Graphical and end-user tools provide facile data collection and visualization.



Component **PAPI**



PAPI High Level Calls

1. PAPI_num_counters()
   - get the number of hardware counters available on the system
2. PAPI_flips(float *rtime, float *ptime, long long *flpins, float *mflips)
   - simplified call to get Mflips/s (floating point instruction rate), real and processor time
3. PAPI_flops (float *rtime, float *ptime, long long *flpops, float *mflops)
   - simplified call to get Mflops/s (floating point operation rate), real and processor time
4. PAPI_ipc (float *rtime, float *ptime, long long *ins, float *ipc)
   - gets instructions per cycle, real and processor time
5. PAPI_accum_counters (long long *values, int array_len)
   - add current counts to array and reset counters
6. PAPI_read_counters (long long *values, int array_len)
   - copy current counts to array and reset counters
7. PAPI_start_counters (int *events, int array_len)
   - start counting hardware events
8. PAPI_stop_counters (long long *values, int array_len)
   - stop counters and return current counts

PAPI Low Level Example

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];

/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);

/* Start the counters */
retval = PAPI_start (EventSet);
do_work();  /* What we want to monitor*/
/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);
```

PAPI Preset Events

- Preset Events
  - Standard set of over 100 events for application performance tuning
  - No standardization of the exact definition
  - Mapped to either single or linear combinations of native events on each platform
  - Use *papi_avail* utility to see what preset events are available on a given platform

PAPI_L2_ICH: Level 2 instruction cache hits
PAPI_L2_ICA: Level 2 instruction cache accesses
PAPI_L2_ICR: Level 2 instruction cache reads
PAPI_L2_ICW: Level 2 instruction cache writes
PAPI_L2_ICM: Level 2 instruction cache misses

PAPI_L2_TCH: Level 2 total cache hits
PAPI_L2_TCA: Level 2 total cache accesses
PAPI_L2_TCR: Level 2 total cache reads
PAPI_L2_TCW: Level 2 total cache writes
PAPI_L2_TCM: Level 2 cache misses

PAPI_L2_LDM: Level 2 load misses
PAPI_L2_STM: Level 2 store misses

Level 3 Cache

PAPI_L3_DCH: Level 3 data cache hits
PAPI_L3_DCA: Level 3 data cache accesses
PAPI_L3_DCR: Level 3 data cache reads
PAPI_L3_DCW: Level 3 data cache writes
PAPI_L3_DCM: Level 3 data cache misses

PAPI_L3_ICH: Level 3 instruction cache hits
PAPI_L3_ICA: Level 3 instruction cache accesses
PAPI_L3_ICR: Level 3 instruction cache reads
PAPI_L3_ICW: Level 3 instruction cache writes
PAPI_L3_ICM: Level 3 instruction cache misses

PAPI_L3_TCH: Level 3 total cache hits
PAPI_L3_TCA: Level 3 total cache accesses
PAPI_L3_TCR: Level 3 total cache reads
PAPI_L3_TCW: Level 3 total cache writes
PAPI_L3_TCM: Level 3 cache misses

PAPI_L3_LDM: Level 3 load misses
PAPI_L3_STM: Level 3 store misses

Cache Sharing

PAPI_CA_SNP: Requests for a snoop
PAPI_CA_SHR: Requests for exclusive access to shared cache line
PAPI_CA_CLN: Requests for exclusive access to clean cache line
PAPI_CA_INV: Requests for cache line invalidation
PAPI_CA_ITV: Requests for cache line intervention

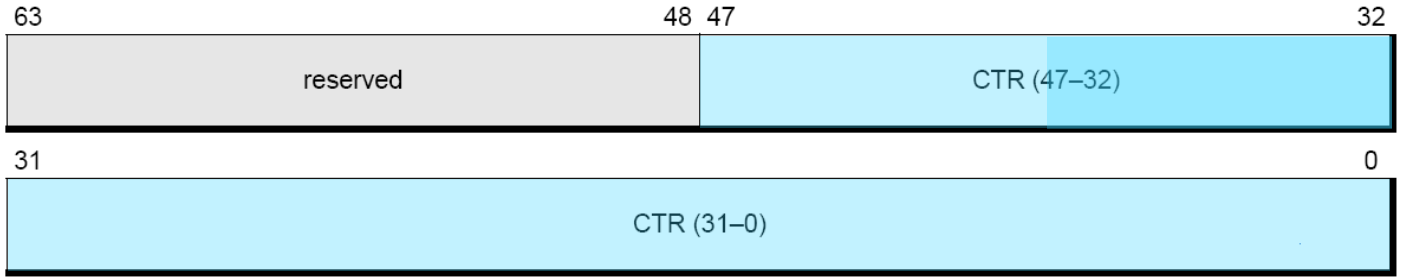
PAPI Native Events

- Native Events
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use *papi_native_avail* utility to see all available native events
- Use *papi_event_chooser* utility to select a compatible set of events

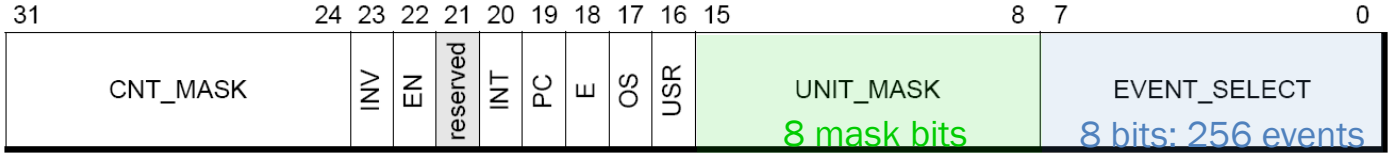
```
PRESET,
  PAPI_L1_DCH,
  DERIVED_SUB,
  DATA_CACHE_ACCESSES,
  DATA_CACHE_MISSES
```

```
/* 19 */{.pme_name =
"DATA_CACHE_REFILLS_FROM_SYSTEM",
    .pme_code  = 0x43,
    .pme_desc  = "Data Cache Refills from
       the northbridge",
    .pme_flags = PFMLIB_AMD64_UMASK_COMBO,
    .pme_numasks = 6,
    .pme_umasks  = {
       { .pme_uname = "INVALID",
         .pme_udesc = "Invalid",
         .pme_ucode = 0x01,
       },
       { .pme_uname = "SHARED",
         .pme_udesc = "Shared",
         .pme_ucode = 0x02,
       },
       { .pme_uname = "EXCLUSIVE",
         .pme_udesc = "Exclusive",
         .pme_ucode = 0x04,
       },
       { .pme_uname = "OWNED",
         .pme_udesc = "Owned",
         .pme_ucode = 0x08,
       },
       { .pme_uname = "MODIFIED",
         .pme_udesc = "Modified",
         .pme_ucode = 0x10,
       },
       { .pme_uname = "ALL",
         .pme_udesc = "All sub-events",
         .pme_ucode = 0x1F,
       },
    },
},
```

How is Implemented?



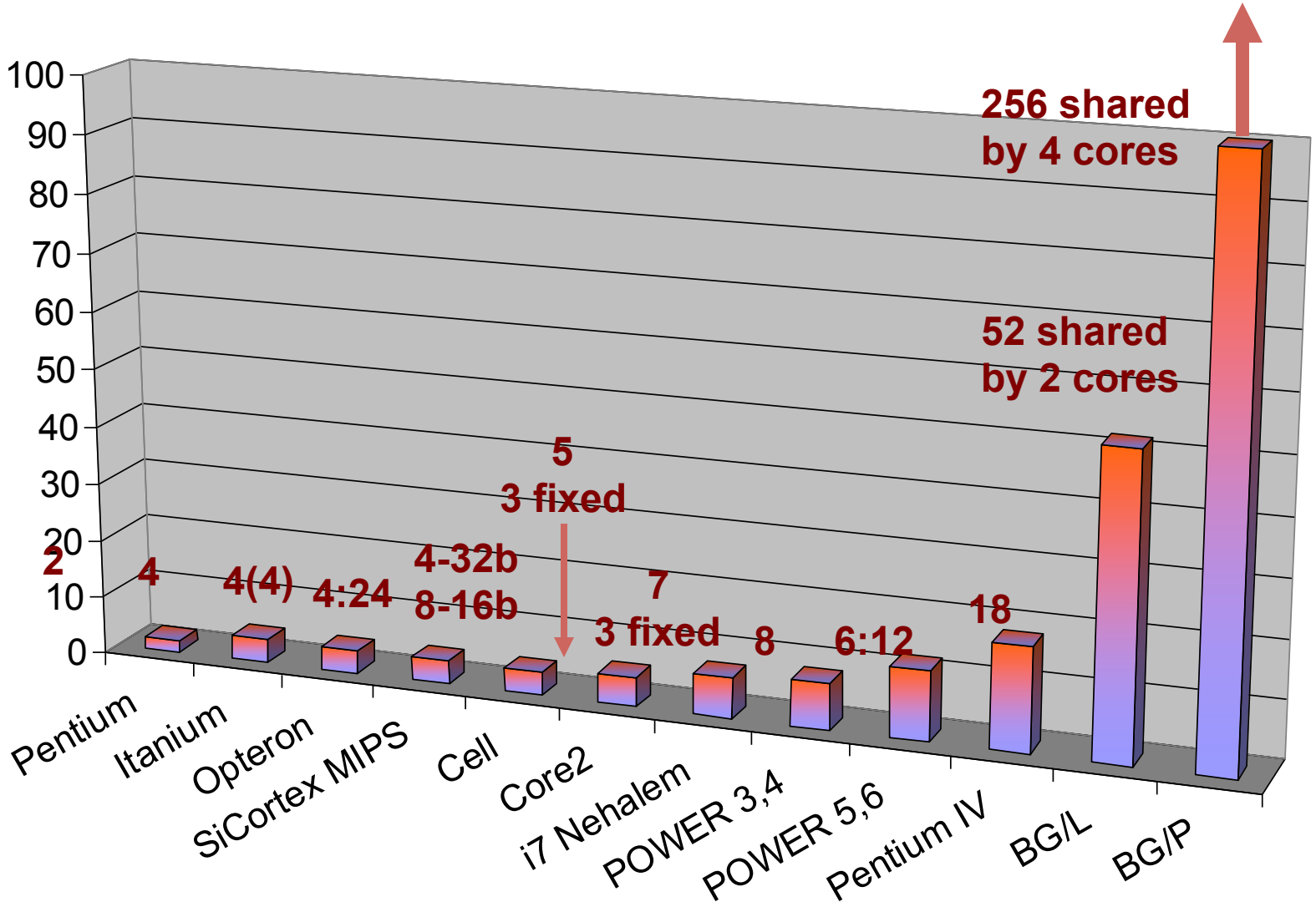
PMD: AMD, Intel



PMC: Intel Pentium II, III, M, Core, i7; AMD Athlon, Opteron

- 2^32 ~= 4 billion ~= 2 seconds @ 2 GHz
- 2^40 ~= 500 seconds
- 2^48 ~= 128,000 seconds ~= 2000 minutes ~= 36 hours
- 2^64 ~= 96,000 days ~= 260 years

How many counters does it take?



Ivy Bridge Counters

- 3 Fixed Function Counters
  - Unhalted Core Cycles
  - Unhalted Reference Cycles
  - Instructions Retired
- 8 Programmable Counters
  - unless you're Hyperthreading (4 per thread)
  - or using an NMI watchdog timer (3 per thread)
- 4 Uncore Counters
  - chip wide; not core specific
  - unified cache measurement (L3)
  - shared resources

Useful PAPI Utilities

- *papi_cost*
- *papi_avail*
- *papi_native_avail*
- *papi_event_chooser*
- *papi_command_line*

PAPI Utilities: *papi_cost*

```
$ utils/papi_cost -h
This is the PAPI cost program.
It computes min / max / mean / std. deviation for PAPI start/stop
  pairs and for PAPI reads.  Usage:

    cost [options] [parameters]
    cost TESTS_QUIET

Options:

  -b BINS       set the number of bins for the graphical
                distribution of costs. Default: 100
  -d            show a graphical distribution of costs
  -h            print this help message
  -s            show number of iterations above the first
                10 std deviations
  -t THRESHOLD  set the threshold for the number of
                iterations. Default: 100,000
```

PAPI Utilities: *papi_cost*

```
$ utils/papi_cost
Cost of execution for PAPI start/stop and PAPI read.
This test takes a while. Please be patient...
Performing start/stop test...

Total cost for PAPI_start/stop(2 counters) over 1000000 iterations
min cycles    : 63
max cycles    : 17991
mean cycles   : 69.000000
std deviation: 34.035263
 Performing start/stop test...

Performing read test...

Total cost for PAPI_read(2 counters) over 1000000 iterations
min cycles    : 288
max cycles    : 102429
mean cycles   : 301.000000
std deviation: 144.694053
 cost.c                                    PASSED
```

PAPI Utilities: *papi_cost*

```
Cost distribution profile

      63:**************************** 999969 counts ****************************
     153:
     243:

     [...]
    1683:
    1773:
    1863:
    1953:********************
    2043:
    2133:********************
    2223:
    2313:
    2403:********************
    2493:********************
    2583:****************************************
    2673:****************************************
    2763:************************************************************************************************
    2853:****************************************
    2943:
    3033:********************
    3123:****************************************
    3213:****************************************
    3303:
    3393:
    3483:
    3573:
    3663:********************
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail -h
Usage: utils/papi_avail [options]
Options:

General command options:
        -a, --avail   Display only available preset events
        -d, --detail  Display detailed information about all preset events
        -e EVENTNAME  Display detail information about specified preset or native event
        -h, --help    Print this help message

Event filtering options:
   --br          Display branch related PAPI preset events
   --cache       Display cache related PAPI preset events
   --cnd         Display conditional PAPI preset events
   --fp          Display Floating Point related PAPI preset events
   --ins         Display instruction related PAPI preset events
   --idl         Display Stalled or Idle PAPI preset events
   --l1          Display level 1 cache related PAPI preset events
   --l2          Display level 2 cache related PAPI preset events
   --l3          Display level 3 cache related PAPI preset events
   --mem         Display memory related PAPI preset events
   --msc         Display miscellaneous PAPI preset events
   --tlb         Display Translation Lookaside Buffer PAPI preset events

This program provides information about PAPI preset and native events.
PAPI preset event filters can be combined in a logical OR.
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.1.0.2
Vendor string and code   : AuthenticAMD (2)
Model string and code    : Six-Core AMD Opteron(tm) Processor 8439 SE (8)
CPU Revision             : 0.000000
CPUID Info               : Family: 16  Model: 8  Stepping: 0
CPU Max Megahertz        : 2812
CPU Min Megahertz        : 2812
Hdw Threads per core     : 1
Cores per Socket         : 6
NUMA Nodes               : 8
CPUs per Node            : 6
Total CPUs               : 48
Running in a VM          : no
Number Hardware Counters : 4
Max Multiplex Counters   : 64
--------------------------------------------------------------------------------

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
[...]
PAPI_FP_OPS  0x80000066  Yes   No   Floating point operations (Counts speculative adds and
   multiplies. Variable and higher than theoretical.)
[...]
PAPI_REF_CYC 0x8000006b  No    No   Reference clock cycles
--------------------------------------------------------------------------------
Of 108 possible events, 40 are available, of which 8 are derived.

avail.c                                  PASSED
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail -a
Available events and hardware information.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  No   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
[…]
PAPI_DP_OPS  0x80000068  No   Floating point operations; optimized to count
  scaled double precision vector operations
--------------------------------------------------------------------------------
Of 40 available events, 8 are derived.

avail.c                                      PASSED
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail -e PAPI_FP_OPS
Available events and hardware information.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------

Event name:                     PAPI_FP_OPS
Event Code:                     0x80000066
Number of Native Events:        1
Short Description:             |FP operations|
Long Description:              |Floating point operations|
Developer's Notes:             ||
Derived Type:                  |NOT_DERIVED|
Postfix Processing String:     ||
 Native Code[0]: 0x4000001d |
  RETIRED_SSE_OPERATIONS:SINGLE_ADD_SUB_OPS:SINGLE_MUL_OPS:DOUBLE_ADD_SUB_O
  PS:DOUBLE_MUL_OPS:OP_TYPE|
 Number of Register Values: 0
 Native Event Description: |Retired SSE Operations, masks:Single precision
  add/subtract ops,Single precision multiply ops,Double precision add/
  subtract ops,Double precision multiply ops,Op type: 0=uops. 1=FLOPS|

--------------------------------------------------------------------------------
avail.c                                     PASSED
```

PAPI Utilities: *papi_native_avail*

```
UNIX> utils/papi_native_avail
Available native events and hardware information.
--------------------------------------------------------------------------------

[...]
================================================================================
 Native Events in Component: perf_events
================================================================================
| perf::PERF_COUNT_HW_CPU_CYCLES                                               |
|            PERF_COUNT_HW_CPU_CYCLES                                          |
--------------------------------------------------------------------------------
| perf::CYCLES                                                                 |
|            PERF_COUNT_HW_CPU_CYCLES                                          |
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------
| perf::PERF_COUNT_SW_PAGE_FAULTS                                              |
|            PERF_COUNT_SW_PAGE_FAULTS                                         |
--------------------------------------------------------------------------------
| perf::PERF_COUNT_SW_CONTEXT_SWITCHES                                         |
|            PERF_COUNT_SW_CONTEXT_SWITCHES                                    |
--------------------------------------------------------------------------------
[...]
```

PAPI Utilities: *papi_native_avail*

```
--------------------------------------------------------------------------------
| DISPATCHED_FPU                                                               |
|            Dispatched FPU Operations                                         |
|     :OPS_ADD                                                                 |
|            Add pipe ops excluding load ops and SSE move ops                  |
|     :OPS_MULTIPLY                                                            |
|            Multiply pipe ops excluding load ops and SSE move ops             |
|     :OPS_STORE                                                               |
|            Store pipe ops excluding load ops and SSE move ops                |
|     :OPS_ADD_PIPE_LOAD_OPS                                                   |
|            Add pipe load ops and SSE move ops                                |
|     :OPS_MULTIPLY_PIPE_LOAD_OPS                                              |
|            Multiply pipe load ops and SSE move ops                           |
|     :OPS_STORE_PIPE_LOAD_OPS                                                 |
|            Store pipe load ops and SSE move ops                              |
|     :ALL   All sub-events selected                                           |
|     :e=0   edge level                                                        |
|     :i=0   invert                                                            |
|     :c=0   counter-mask in range [0-255]                                     |
|     :g=0   measure in guest                                                  |
|     :u=0   monitor at user level                                             |
|     :k=0   monitor at kernel level                                           |
|     :h=0   monitor at hypervisor level                                       |
--------------------------------------------------------------------------------
```

PAPI Utilities: *papi_native_avail*

```
UNIX> utils/papi_native_avail -e DATA_CACHE_REFILLS
Available native events and hardware information.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------
Event name:                      DATA_CACHE_REFILLS
Description:                    |Data Cache Refills from L2 or Northbridge|

Unit Masks:
 Mask Info:                     |:SYSTEM|Refill from the Northbridge|
 Mask Info:                     |:L2_SHARED|Shared-state line from L2|
 Mask Info:                     |:L2_EXCLUSIVE|Exclusive-state line from L2|
 Mask Info:                     |:L2_OWNED|Owned-state line from L2|
 Mask Info:                     |:L2_MODIFIED|Modified-state line from L2|
 Mask Info:                     |:ALL|All sub-events selected|
 Mask Info:                     |:e=0|edge level|
 Mask Info:                     |:i=0|invert|
 Mask Info:                     |:c=0|counter-mask in range [0-255]|
 Mask Info:                     |:g=0|measure in guest|
 Mask Info:                     |:u=0|monitor at user level|
 Mask Info:                     |:k=0|monitor at kernel level|
 Mask Info:                     |:h=0|monitor at hypervisor level|
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser
Usage: eventChooser NATIVE|PRESET evt1 evt2 ...
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser PRESET PAPI_FP_OPS
Event Chooser: Available events which can be added with given events.
-------------------------------------------------------------------------
[…]
-------------------------------------------------------------------------
    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
[…]
PAPI_L1_DCA  0x80000040  No   Level 1 data cache accesses
PAPI_L2_DCR  0x80000044  No   Level 2 data cache reads
PAPI_L2_DCW  0x80000047  No   Level 2 data cache writes
PAPI_L1_ICA  0x8000004c  No   Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  No   Level 2 instruction cache accesses
PAPI_L2_TCA  0x80000059  No   Level 2 total cache accesses
PAPI_L2_TCW  0x8000005f  No   Level 2 total cache writes
PAPI_FML_INS 0x80000061  No   Floating point multiply instructions
PAPI_FDV_INS 0x80000063  No   Floating point divide instructions
-------------------------------------------------------------------------
Total events reported: 34
event_chooser.c                          PASSED
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser PRESET PAPI_FP_OPS PAPI_L1_DCM
Event Chooser: Available events which can be added with given events.
-------------------------------------------------------------------------
[…]
-------------------------------------------------------------------------

    Name        Code    Deriv Description (Note)
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_TOT_CYC 0x8000003b  No   Total cycles
-------------------------------------------------------------------------
Total events reported: 2
event_chooser.c                          PASSED
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser NATIVE RESOURCE_STALLS:LD_ST X87_OPS_RETIRED
    INSTRUCTIONS_RETIRED
[…]
-------------------------------------------------------------------------
UNHALTED_CORE_CYCLES    0x40000000
 |count core clock cycles whenever the clock signal on the specific core is running (not
   halted). Alias to event CPU_CLK_UNHALTED:CORE_P|
 |Register Value[0]: 0x20003       Event Selector|
 |Register Value[1]: 0x3c          Event Code|
-------------------------------------------------------------------------
UNHALTED_REFERENCE_CYCLES       0x40000002
 |Unhalted reference cycles. Alias to event CPU_CLK_UNHALTED:REF|
 |Register Value[0]: 0x40000       Event Selector|
 |Register Value[1]: 0x13c         Event Code|
-------------------------------------------------------------------------
CPU_CLK_UNHALTED        0x40000028
 |Core cycles when core is not halted|
 |Register Value[0]: 0x60000       Event Selector|
 |Register Value[1]: 0x3c          Event Code|
    0x40001028  :CORE_P  |Core cycles when core is not halted|
    0x40008028  :NO_OTHER  |Bus cycles when core is active and the other is halted|
-------------------------------------------------------------------------
Total events reported: 3
event_chooser.c                         PASSED
```

PAPI Utilities: *papi_command_line*

```
$ papi_command_line PAPI_FP_OPS
Successfully added: PAPI_FP_OPS

PAPI_FP_OPS :   100000000

---------------------------------
Verification: None.
 This utility lets you add events from the command line interface to see if they work.
command_line.c                          PASSED


$ papi_command_line PAPI_FP_OPS PAPI_L1_DCA
Successfully added: PAPI_FP_OPS
Successfully added: PAPI_L1_DCA

PAPI_FP_OPS :   100000000
PAPI_L1_DCA :   120034404

---------------------------------
Verification: None.
 This utility lets you add events from the command line interface to see if they work.
command_line.c                          PASSED
```

Performance Measurement Categories

- Efficiency
  - Instructions per cycle (IPC)
  - Memory bandwidth
- Caches
  - Data cache misses and miss ratio
  - Instruction cache misses and miss ratio
- Translation lookaside buffers (TLB)
  - Data TLB misses and miss ratio
  - Instruction TLB misses and miss ratio
- Control transfers
  - Branch mispredictions
  - Near return mispredictions
- Special cases
  - Unaligned data access
  - Floating point operations
  - Floating point exceptions

The Code

```
#define ROWS 1000    // Number of rows in each matrix
#define COLUMNS 1000     // Number of columns in each matrix
```

```
void classic_matmul()
{
   // Multiply the two matrices
   int i, j, k;
   for (i = 0; i < ROWS; i++) {
      for (j = 0; j < COLUMNS; j++) {

         for (k = 0; k < COLUMNS; k++) {
            sum +=
               matrix_a[i][k] * matrix_b[k][j];
         }
         matrix_c[i][j] = sum;
      }
   }
}
```

```
void interchanged_matmul()
{
   // Multiply the two matrices
   int i, j, k;
   for (i = 0; i < ROWS; i++) {
      for (k = 0; k < COLUMNS; k++) {
         for (j = 0; j < COLUMNS; j++) {
            matrix_c[i][j] +=
               matrix_a[i][k] * matrix_b[k][j];
         }
      }
   }
}
```

// Note that the nesting of the innermost
// loops has been changed. The index variables j
// and k change the most frequently and the access
// pattern through the operand matrices is sequential
// using a small stride (one.) This change improves
// access to memory data through the data cache.
// Data translation lookaside buffer (DTLB) behavior
// is also improved.

Performance Data

IPC – instructions per cycle

- A measure of instruction level parallelism
- An indicator of code efficiency

PAPI_ipc

```
retval = PAPI_ipc(&realtime, &processtime, &start_ins, &ipc);
classic_matmul();
retval = PAPI_ipc(&realtime, &processtime, &end_ins, &ipc);
retval = PAPI_stop_counters(NULL, 0));
```

PAPI High Level

```
int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

realtime[0] = PAPI_get_real_usec();
retval = PAPI_start_counters(events, 2);
classic_matmul();
retval = PAPI_stop_counters(cvalues, 2);
realtime[1] = PAPI_get_real_usec();
```

PAPI Low Level

```
int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

retval = PAPI_library_init (PAPI_VER_CURRENT);
retval = PAPI_create_eventset(&EventSet);
retval = PAPI_add_events(EventSet, events, 2);
realtime[0] = PAPI_get_real_usec();
retval = PAPI_start(EventSet);
classic_matmul();
retval = PAPI_stop(EventSet, cvalues);
realtime[1] = PAPI_get_real_usec();
```

IPC – instructions per cycle

```
Measurement                          Classic mat_mul     Reordered mat_mul
=============================================================================

PAPI_IPC Test (PAPI_ipc)
Real time                             13.6093 sec       2.9796 sec
Processor time                        13.5359 sec       2.9556 sec
IPC                                    0.3697           1.6936
Instructions                           9007035063       9009011383

High Level IPC Test (PAPI_{start,stop}_counters)
Real time                             13.6106 sec       2.9762 sec
IPC                                    0.3697           1.6939
PAPI_TOT_CYC                          24362605525       5318626915
PAPI_TOT_INS                           9007034503       9009011245

Low  Level IPC Test (PAPI low level calls)
Real time                             13.6113 sec       2.9772 sec
IPC                                    0.3697           1.6933
PAPI_TOT_CYC                          24362750167       5320395138
PAPI_TOT_INS                           9007034381       9009011130
```

- All three PAPI methods consistent
- Roughly 460% improvement in reordered code

Data Cache Access

Data Cache Misses can be considered in 3 categories:

- Compulsory: Occurs on first reference to a data item.
  - Prefetching can hide latency
- Capacity: Occurs regardless of cache associativity or line size, solely due to the finite size of the cache
  - Larger caches
  - Improved data locality (tiling / fusion) – effectively lowers working set
- Conflict: Misses that could have been avoided with a fully-associative cache
  - Increase cache associativity
  - Data layout; memory access patterns

L1 Data Cache Access

```
Measurement                              Classic mat_mul    Reordered mat_mul
=============================================================================

DATA_CACHE_ACCESSES                       2002807841       3008528961
DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED
                                           205968263         60716301
DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED
                                            61970925          1950282
---------------------
PAPI_L1_DCA                               2002808034       3008528895
PAPI_L1_DCM                                268010587         62680818
PAPI_TOT_INS                              9007034381       9009011130

Data Cache Request Rate                   0.2224 req/inst    0.3339 req/inst
Data Cache Miss Rate                      0.0298 miss/inst   0.0070 miss/inst
Data Cache Miss Ratio                     0.1338 miss/req    0.0208 miss/req
```

- Two techniques
  - Using native events
  - Using PAPI presets only
- ~50% more requests from reordered code
- 1/4 as many misses per instruction
- 1/6 as many misses per request

L1 Instruction Cache Access

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_L1_ICR	3014322225	3014205662
INSTRUCTION_CACHE_REFILLS_FROM_L2	22	3
INSTRUCTION_CACHE_REFILLS_FROM_SYSTEM	73	36
PAPI_L1_ICR	3014322033	3014205070
PAPI_L1_ICM	60	44
Instr Cache Request Rate	0.3347 req/inst	0.3346 req/inst
Instr Cache Miss Rate	0.0000 miss/inst	0.0000 miss/inst
Instr Cache Miss Ratio	0.0000 miss/req	0.0000 miss/req

- Two techniques, like Data Cache case
  - Using native events
  - Using PAPI presets only
- Small subroutines fit completely in cache
- Virtually no misses; pretty boring

L2 Cache Access

Measurement	Classic mat_mul	Reordered mat_mul
Direct L2 Cache Test		
REQUESTS_TO_L2:INSTRUCTIONS:DATA:TLB_WALK	1057556622	70996294
L2_CACHE_MISS:INSTRUCTIONS:DATA:TLB_WALK	62120093	4167947
L2 Cache Request Rate	0.1174 req/inst	0.0079 req/inst
L2 Cache Miss Rate	0.0069 miss/inst	0.0005 miss/inst
L2 Cache Miss Ratio	0.0587 miss/req	0.0587 miss/req
Indirect L2 Cache Test		
INSTRUCTION_CACHE_REFILLS_FROM_L2	4	0
INSTRUCTION_CACHE_REFILLS_FROM_SYSTEM	30	9
L2_CACHE_MISS:TLB_WALK	260	5438
REQUESTS_TO_L2:TLB_WALK	787632271	803242
DATA_CACHE_REFILLS:L2_SHARED:L2_EXCLUSIVE:L2_OWNED:L2_MODIFIED	205977083	60715886
DATA_CACHE_REFILLS_FROM_SYSTEM:SHARED:EXCLUSIVE:OWNED:MODIFIED	61973057	1950318
L2 Cache Request Rate	0.1172 req/inst	0.0068 req/inst
L2 Cache Miss Rate	0.0069 miss/inst	0.0002 miss/inst
L2 Cache Miss Ratio	0.0587 miss/req	0.0318 miss/req

L2 Cache Access

Measurement	Classic mat_mul	Reordered mat_mul
L2 Cache Request Rate	0.1172 req/inst	0.0068 req/inst
L2 Cache Miss Rate	0.0069 miss/inst	0.0002 miss/inst
L2 Cache Miss Ratio	0.0587 miss/req	0.0318 miss/req
L2 Instr Fraction	0.0000	0.0000
L2 Data Fraction	0.2538	0.9873
L2 TLB Fraction	0.7462	0.0127

- L2 cache is unified on Opteron
- Two techniques:
  - First is coarser grained
  - Second provides more detail but requires 7 events (two passes)
  - No major differences for this code
- L2 requests and misses down dramatically in reordered code
  - Recall, memory accesses are up by 50%
- Almost all (98+%) L2 access are for data in reordered code

DTLB Access

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_L1_DCA	2002809207	3008530341
L1_DTLB_MISS_AND_L2_DTLB_HIT:ALL	296943120	350824
L1_DTLB_AND_L2_DTLB_MISS:ALL	783208861	785470
PAPI_TOT_INS	9007034381	9009011130
L1 DTLB Request Rate	0.2224 req/inst	0.3339 req/inst
L1 DTLB Miss Rate	0.1199 miss/inst	0.0001 miss/inst
L1 DTLB Miss Ratio	0.5393 miss/req	0.0004 miss/req
L2 DTLB Request Rate	0.1199 req/inst	0.0001 req/inst
L2 DTLB Miss Rate	0.0870 miss/inst	0.0001 miss/inst
L2 DTLB Miss Ratio	0.7251 miss/req	0.6913 miss/req

- L1 Data Cache Access == DTLB Access
- More L1 accesses in improved code
- Dramatically fewer misses
- TLB misses can limit fast matrix multiply

ITLB Access

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_L1_ICR	3014320811	3014204576
L1_ITLB_MISS_AND_L2_ITLB_HIT	4	1
L1_ITLB_MISS_AND_L2_ITLB_MISS:ALL	9	6
L1 ITLB Request Rate	0.3347 req/inst	0.3346 req/inst
L1 ITLB Miss Rate	0.0000 miss/inst	0.0000 miss/inst
L1 ITLB Miss Ratio	0.0000 miss/req	0.0000 miss/req
L2 ITLB Request Rate	0.0000 req/inst	0.0000 req/inst
L2 ITLB Miss Rate	0.0000 miss/inst	0.0000 miss/inst
L2 ITLB Miss Ratio	0.6923 miss/req	0.8571 miss/req

- See DTLB…
- L1 Instruction Cache Reads == ITLB Access
- Boring…but useful in identifying code layout problems

Branching

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_BR_INS	1001028240	1001006987
PAPI_BR_MSP	1028256	1006984
PAPI_BR_TKN	1000027233	1000005980
Branch Rate	0.1111 br/inst	0.1111 br/inst
Branch Miss Rate	0.0001 miss/inst	0.0001 miss/inst
Branch Miss Ratio	0.0010 miss/br	0.0010 miss/br
Branch Taken Rate	0.1110 tkn/inst	0.1110 tkn/inst
Branch Taken Ratio	0.9990 tkn/br	0.9990 tkn/br
Instr / Branch	8.9978 inst/br	8.9999 inst/br

- Uses all PAPI Presets!
- Branch behavior nearly identical in both codes
- Roughly 1 branch every 9 instructions
- 1 miss per 1000 branches (remember ROWS?)
- Branching and branch misses can be reduced with loop unrolling, loop fusion and function in-lining.

Resources

- Intel Developers Guide, Vol 3B (Chapters 18, 19)
  - http://download.intel.com/products/processor/manual/253669.pdf
- AMD BIOS and Kernel Developer Guides
  - http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/
- Performance Analysis Examples
  - http://developer.amd.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf
  - http://developer.amd.com/wordpress/media/2012/10/Introduction_to_CodeAnalyst.pdf
- Performance Tuning of Scientific Applications
  - http://www.amazon.com/Performance-Scientific-Applications-Chapman-Computational/dp/1439815690
  - http://booksgreatchoice.com/getbook/p236074/
    (sign up for a 1 day account for $3.90)

Outline

1. **Part I**
   - Motivation
   - Introduction to Computer Architecture
   - Overview of Performance Analysis techniques
2. **Part II**
   - Introduction to Hardware Counter Events
   - **PAPI**: Access to hardware performance counters
3. **Part III**
   - **HPCToolkit**: Low overhead, full code profiling using hardware counters sampling

Performance Analysis Challenges

- Complex applications present challenges
  - For measurement and analysis
  - For understanding and tuning
- PAPI provides instrumentation based interface to hardware counters
- Easy to use if
  - Code regions of interest are already known, and
  - In small numbers

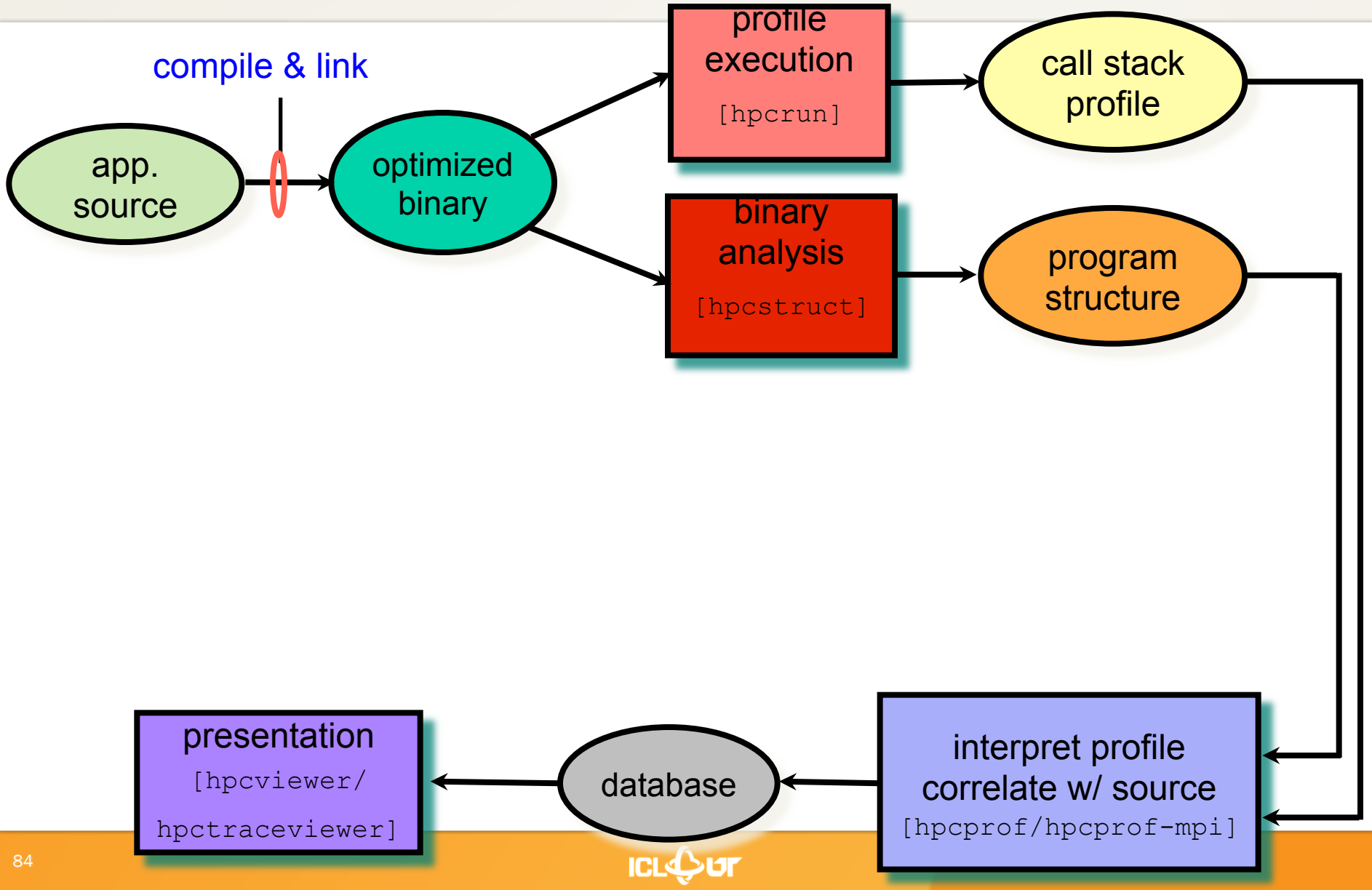
HPCToolkit Overview

- Employs binary-level measurement and analysis
  - Observe fully optimized, dynamically linked executions
  - Support multi-lingual codes with external binary-only libraries
- Uses sampling-based measurement (avoid instrumentation)
  - Controllable overhead
  - Full code coverage, avoid blind spots
  - Enable data collection for large-scale parallelism

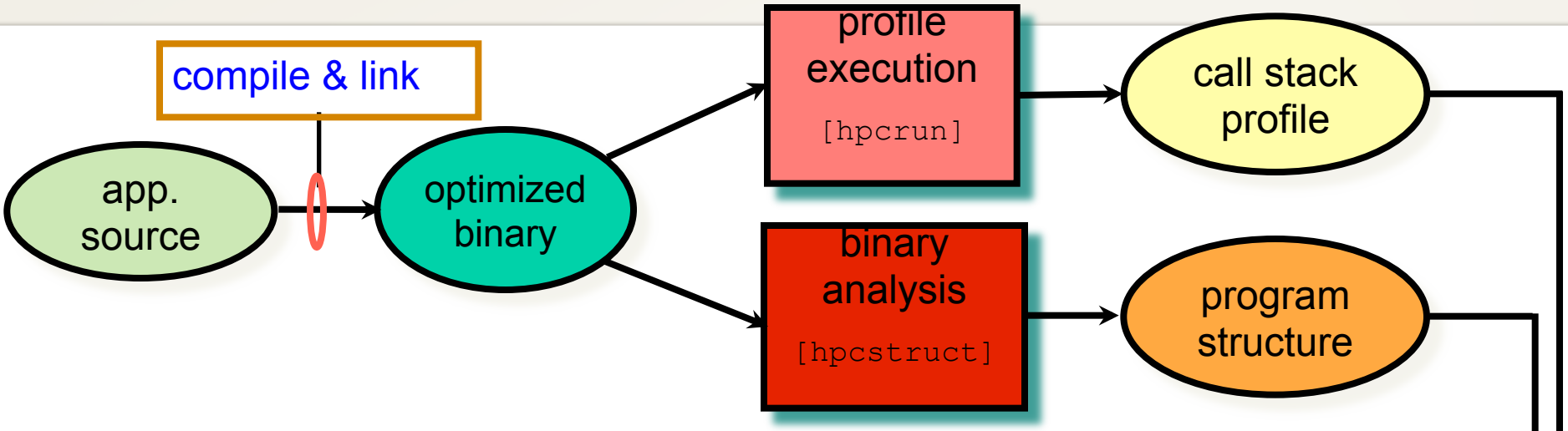
HPCToolkit Overview

- Collects and correlates multiple derived performance metrics
  - Diagnosis typically requires more than one species of metric
- Associates metrics with both static and dynamic context
  - Loop nests, procedures, inlined code, calling context
- Supports top-down performance analysis
  - Natural approach that minimizes burden on developers

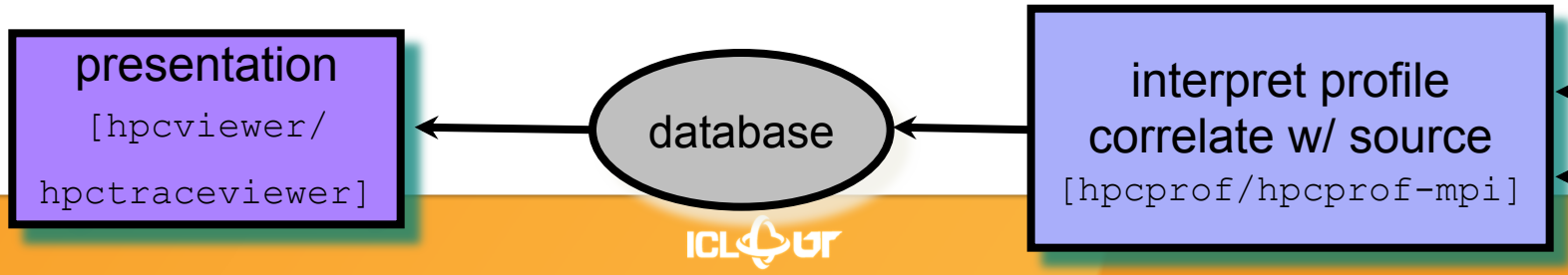
HPCToolkit Workflow



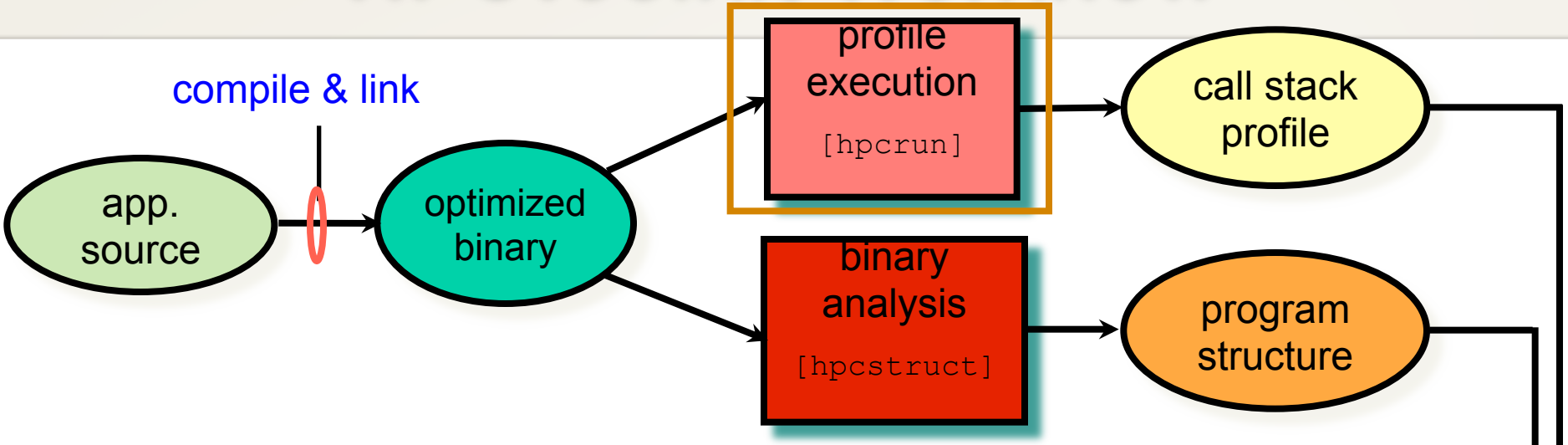
HPCToolkit Workflow



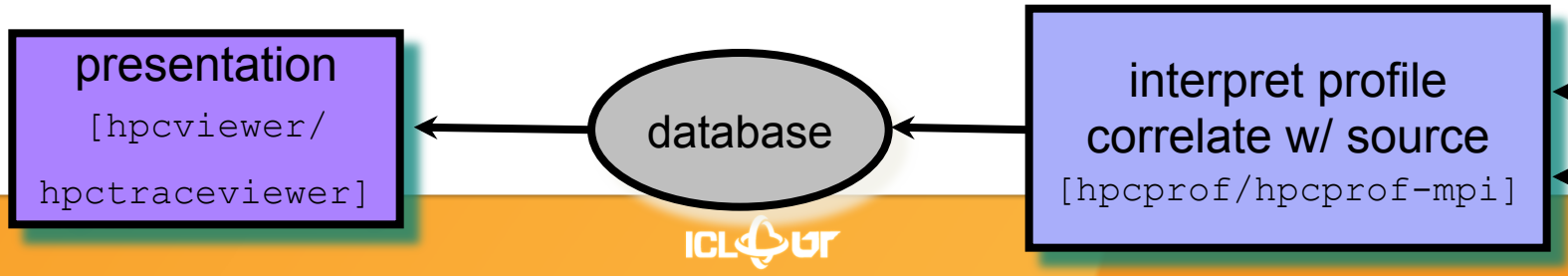
- For dynamically-linked executables on stock Linux
  - compile and link as you usually do: nothing special needed
  - compile code with line mapping information (add flag –g)



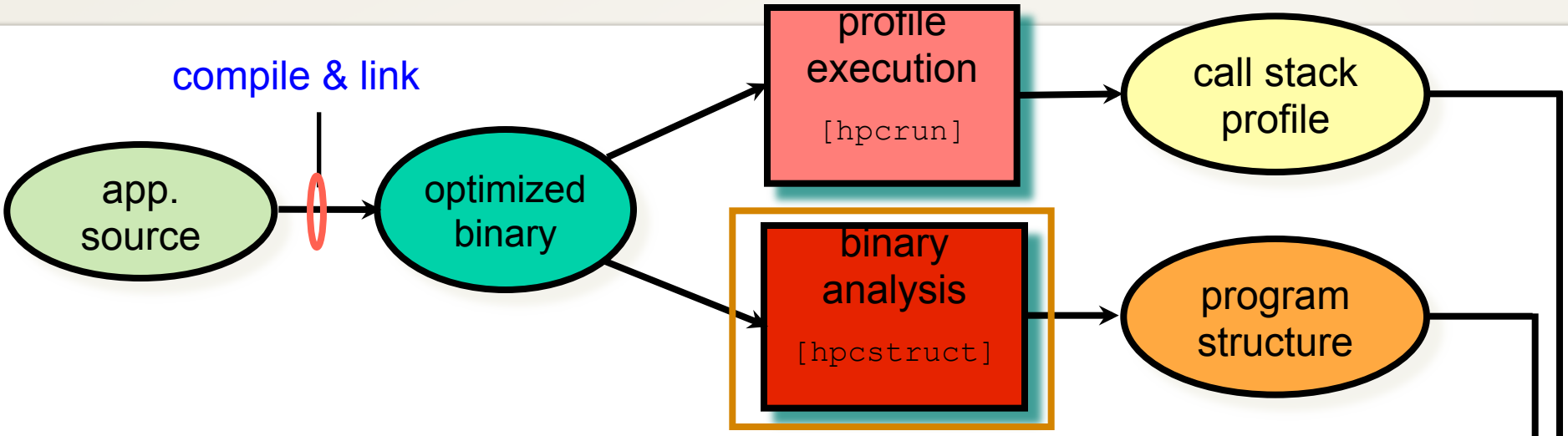
HPCToolkit Workflow



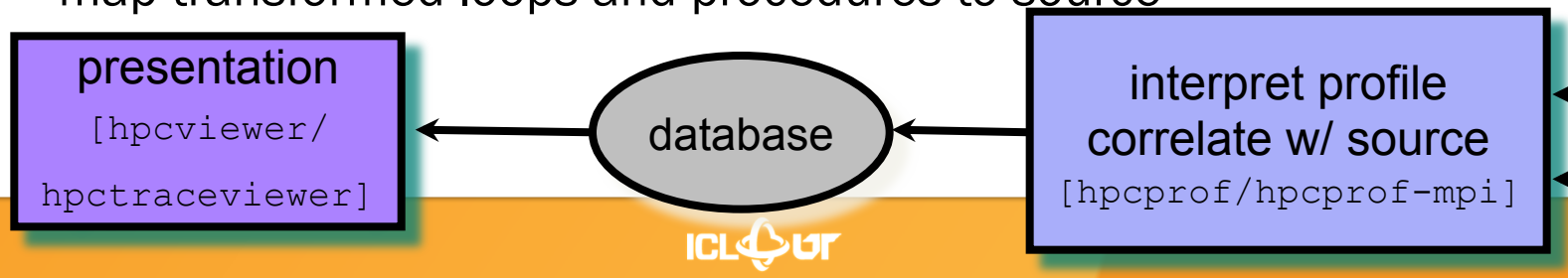
- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with **`hpcrun`** to measure
  - collect statistical call path profiles of events of interest



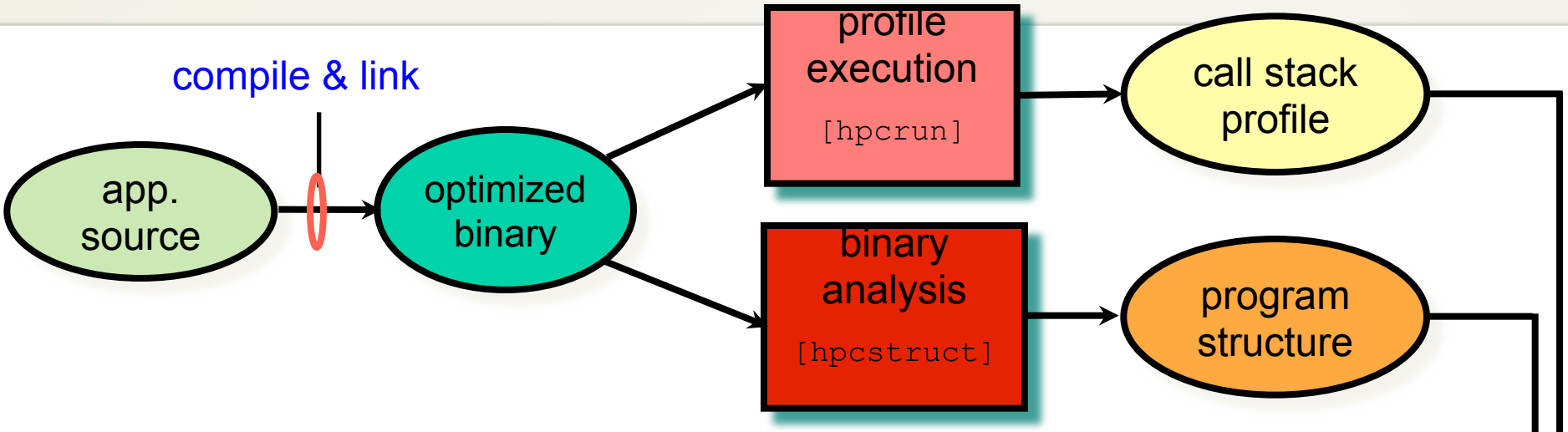
HPCToolkit Workflow



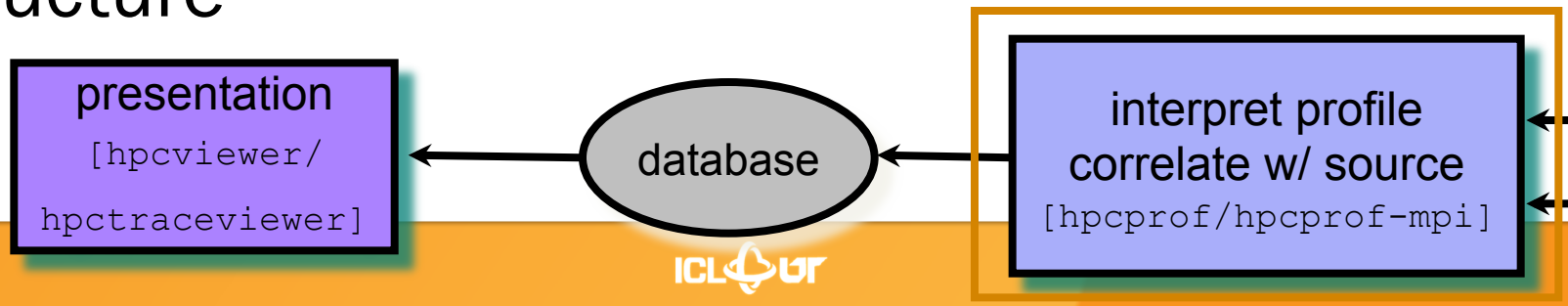
- Analyze binary with **`hpcstruct`**: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source



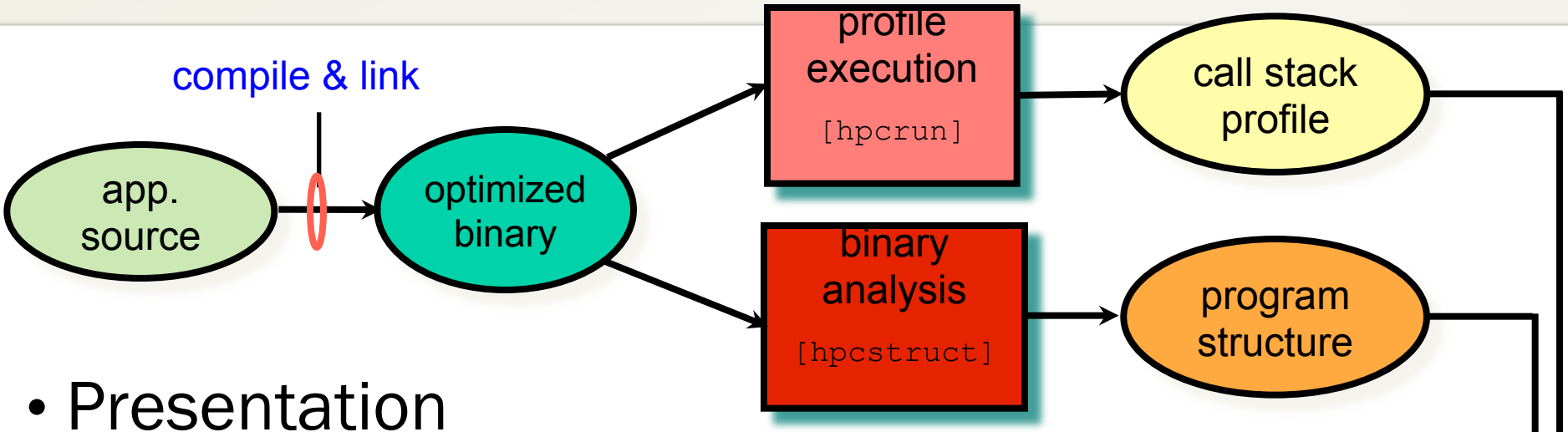
HPCToolkit Workflow



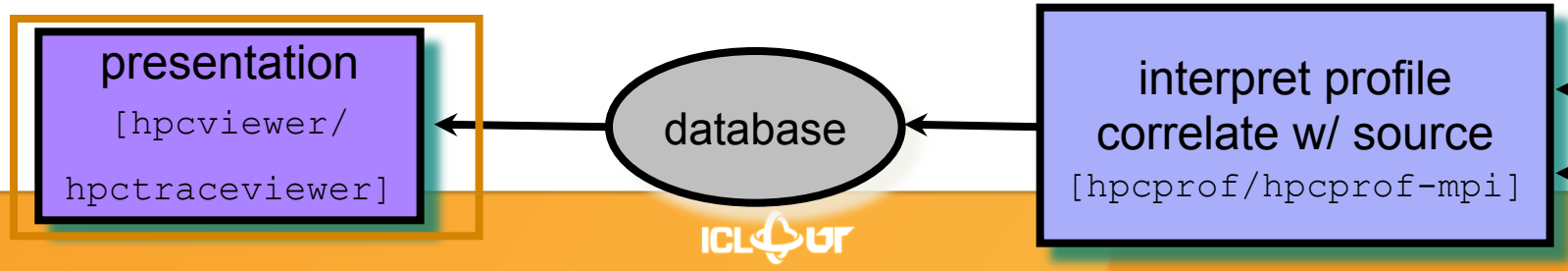
- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure



HPCToolkit Workflow



- Presentation
  - explore performance data from multiple perspectives
    - sort data by metrics to focus on what's important
    - compute derived metrics to help gain insight
      - e.g. scalability losses, waste, IPC, bandwidth



Using HPCToolkit

- Add hpctoolkit's bin directory to your path
  - Download, build and use instructions at http://hpctoolkit.org
  - Installed on newton cluster in "`/home/gmarin/HPCToolkit/`"
- Perhaps adjust your compiler flags for your application
  - most compilers throw away the line map unless -g is on the command line. add -g flag after any optimization flags
- Decide what hardware counters to monitor
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling, or
    - use papi_avail and papi_native_avail
      - you can sample any event listed as "profilable"

Using HPCToolkit

- Profile execution:
  - `hpcrun` –e <event1@period1> [-e <event2@period2> …] <command> [command-arguments]
  - Produces one .hpcrun results file per thread
- Recover program structure
  - `hpcstruct` <command>
  - Produces one .hpcstruct file containing the loop structure of the binary
- Correlate measurements with source code
  - `hpcprof` [–S <hpcstruct_file>] [-M thread] [–o <output_db_name>] <hpcrun_files>
  - Creates performance database
- Use `hpcviewer` to visualize performance results
  - Download `hpcviewer` for your platform from http://code.google.com/p/hpcviewer/downloads/list

Hands-On Demo

```
void compute() {
  int i, j, k, r;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
         C(i,j) += A(i,k) * B(k,j);
}

void main() {
…
  for (i=0 ; i<reps ; ++i) {
     compute();
     if ((i % 2) == 0)
        compute();
  }
…
}
```

Two different call sites to the compute routine

Second call-site executed only half of the time

Hands-On Demo

- Compiled binary is called `matmul`
- Measure `PAPI_TOT_CYC`, `PAPI_TOT_INS`, `PAPI_FP_INS` and `PAPI_LD_INS`

```
$ hpcrun -e PAPI_TOT_CYC@6000000 -e PAPI_TOT_INS@6000000 -e
PAPI_FP_INS@2000000 -e PAPI_LD_INS@3000000 -o hpc_matmul ./matmul 500
```

- Recover program structure

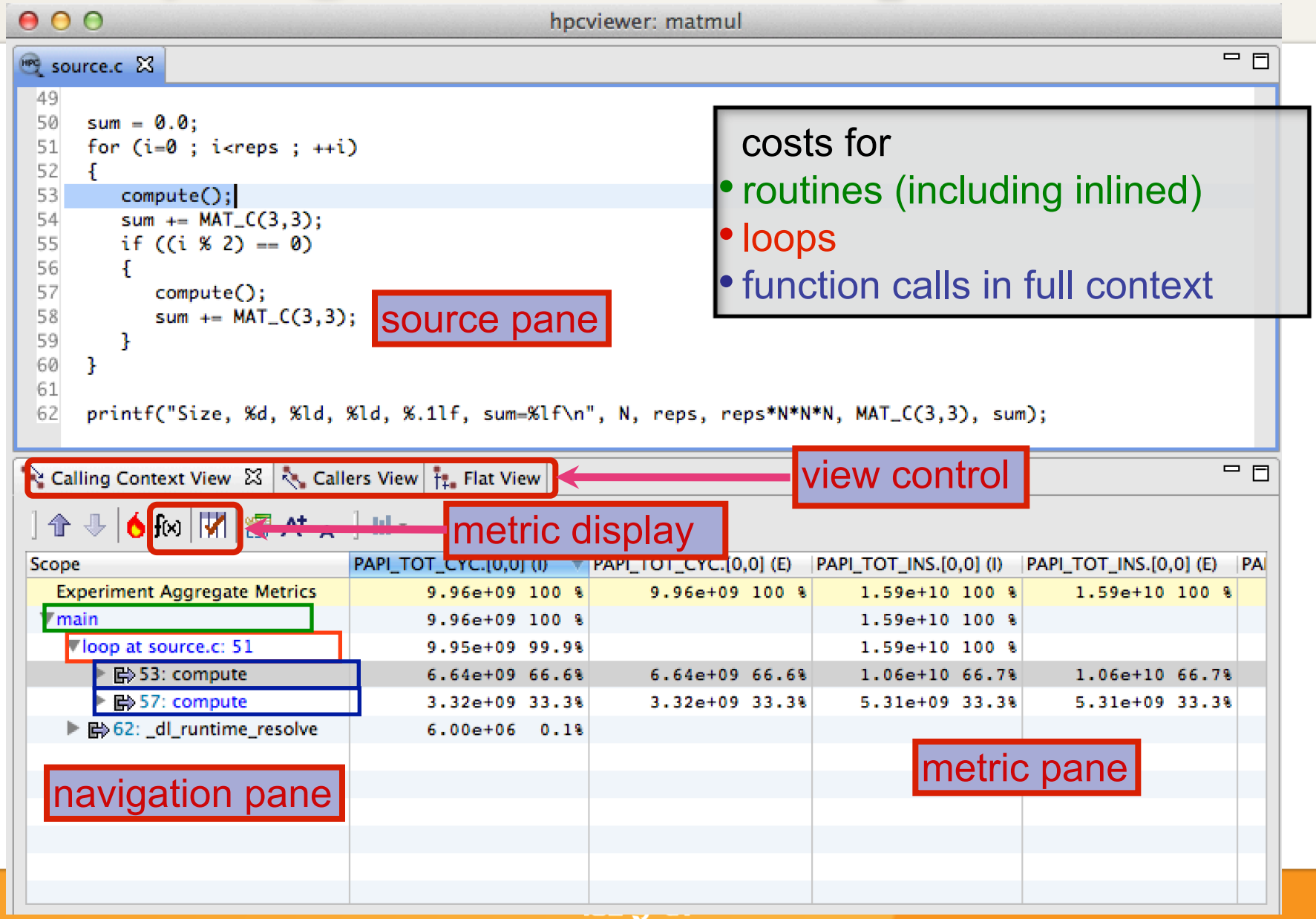
```
$ hpcstruct matmul
```

- Correlate measurements with source code

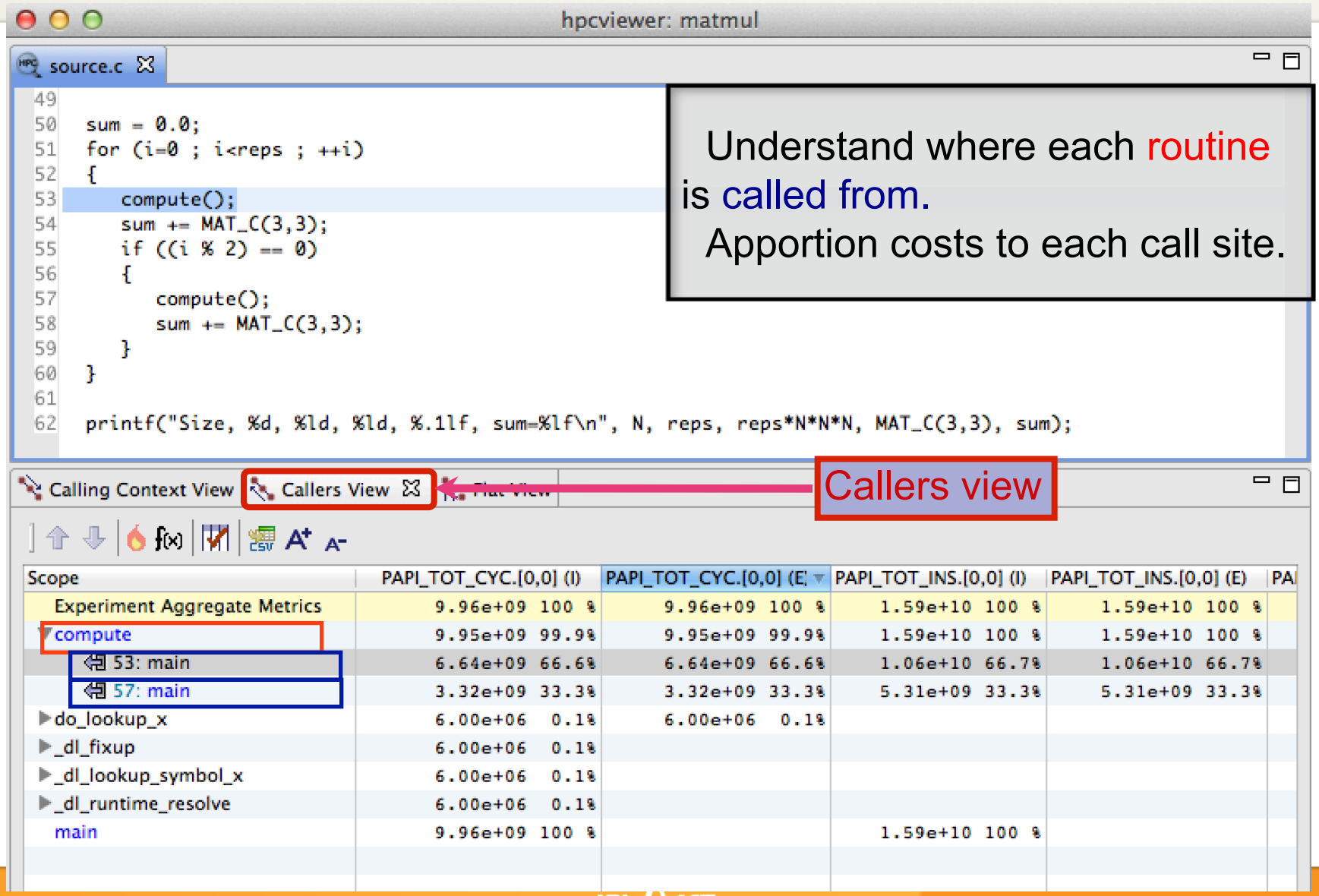
```
$ hpcprof -S matmul.hpcstruct -o db_matmul -M thread
hpc_matmul/matmul-000000-000-*.hpcrun
```

- Copy database to local machine
- Open database in `hpcviewer`

Analyzing results with `hpcviewer`

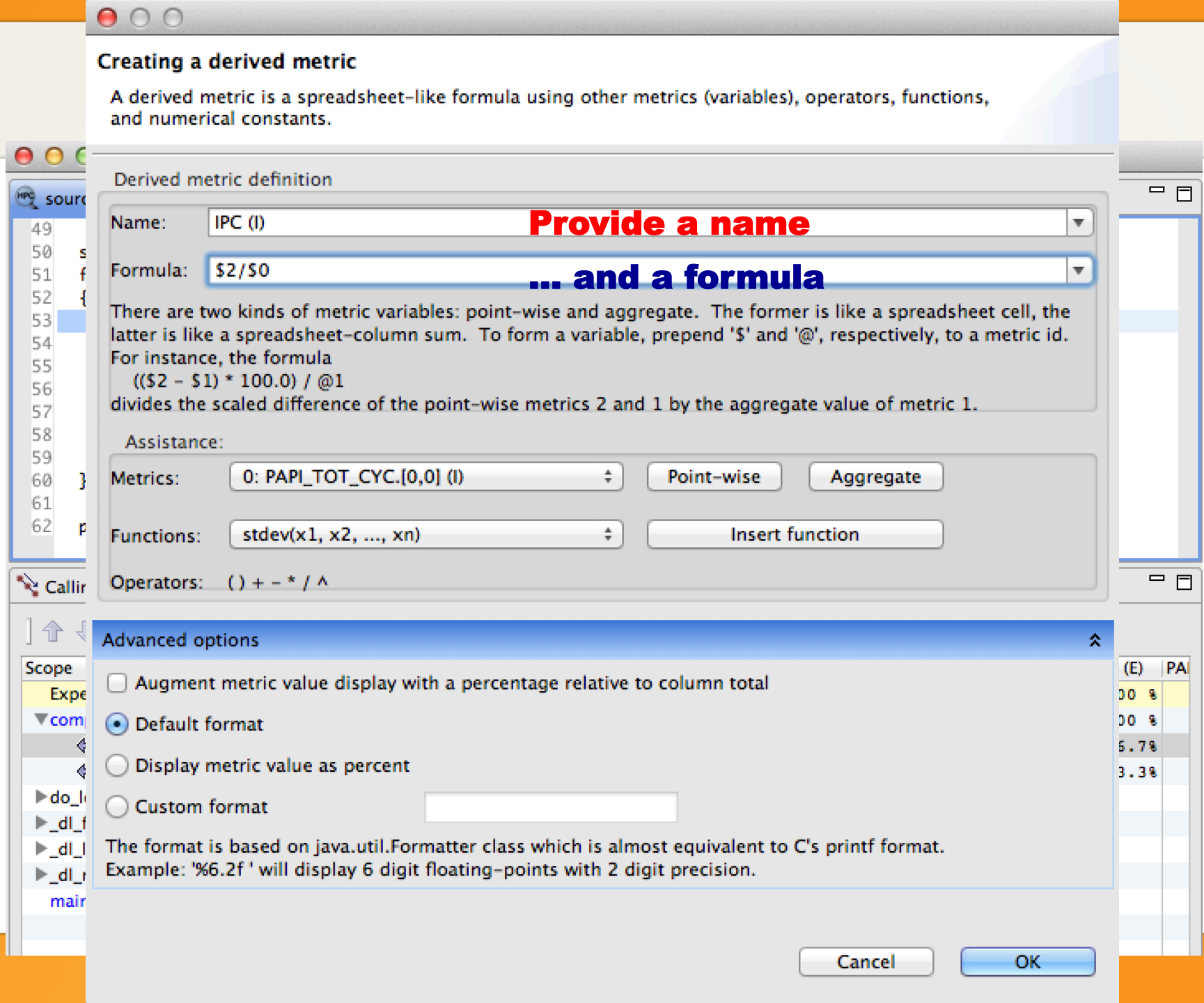


Analyzing results with `hpcviewer`

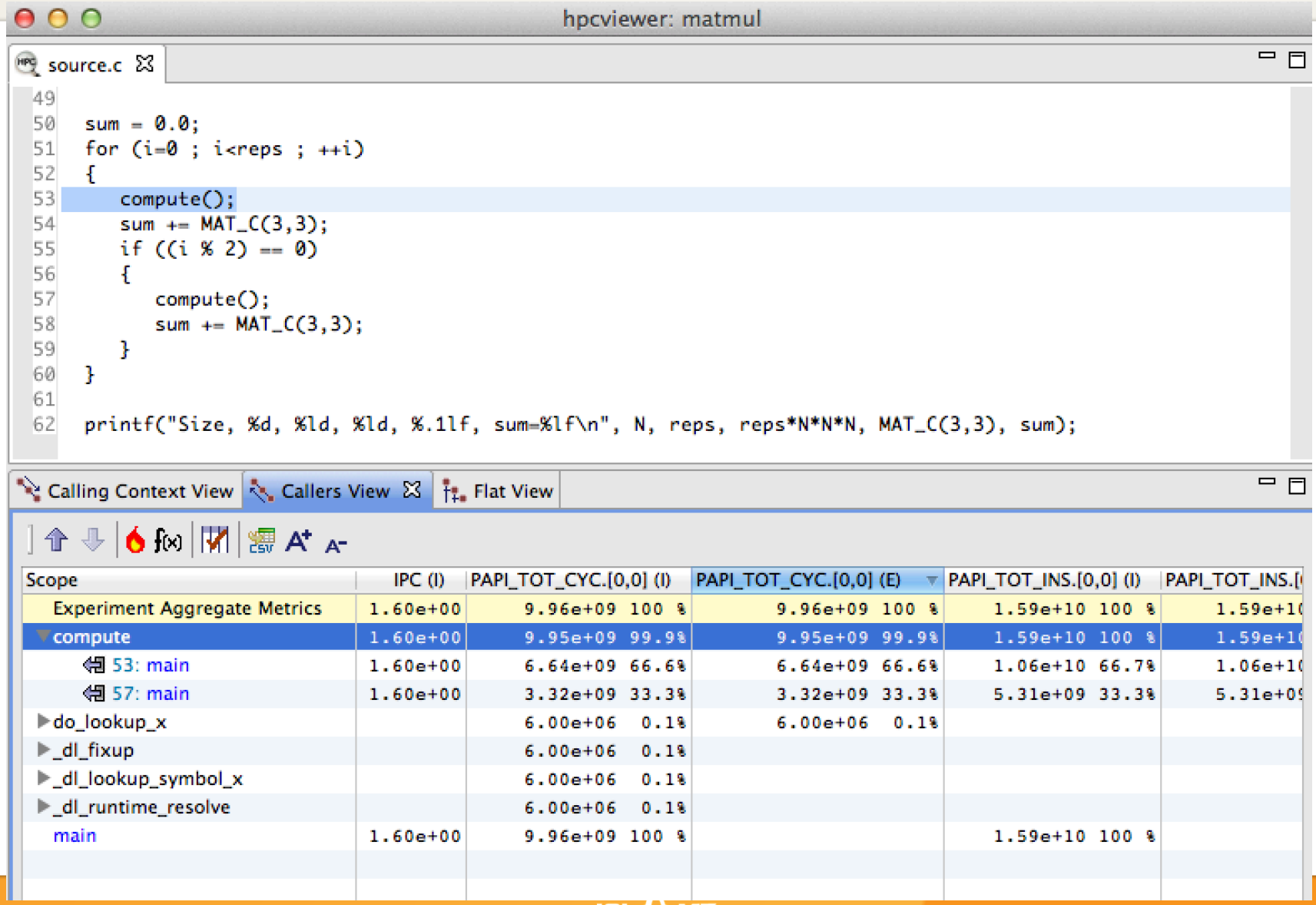

hpcviewer: matmul
source.c
49
50 sum = 0.0;
51 for (i=0 ; i<reps ; ++i)
52 {
53 compute();
54 sum += MAT_C(3,3);
55 if ((i % 2) == 0)
56 {
57 compute();
58 sum += MAT_C(3,3);
59 }
60 }
61
62 printf("Size, %d, %ld, %ld, %.1lf, sum=%lf\n", N, reps, reps*N*N*N, MAT_C(3,3), sum);
Understand where each routine is called from.
Apportion costs to each call site.
Calling Context View
Callers View
Flat View
Callers view
Scope
PAPI_TOT_CYC.[0,0] (I)
PAPI_TOT_CYC.[0,0] (E)
PAPI_TOT_INS.[0,0] (I)
PAPI_TOT_INS.[0,0] (E)
PA
Experiment Aggregate Metrics 9.96e+09 100 % 9.96e+09 100 % 1.59e+10 100 % 1.59e+10 100 %
compute 9.95e+09 99.9% 9.95e+09 99.9% 1.59e+10 100 % 1.59e+10 100 %
53: main 6.64e+09 66.6% 6.64e+09 66.6% 1.06e+10 66.7% 1.06e+10 66.7%
57: main 3.32e+09 33.3% 3.32e+09 33.3% 5.31e+09 33.3% 5.31e+09 33.3%
do_lookup_x 6.00e+06 0.1% 6.00e+06 0.1%
_dl_fixup 6.00e+06 0.1%
_dl_lookup_symbol_x 6.00e+06 0.1%
_dl_runtime_resolve 6.00e+06 0.1%
main 9.96e+09 100 % 1.59e+10 100 %

Principal Views

- Calling context tree view - top-down (down the call chain)
  - associate metrics with each dynamic calling context
  - high-level, hierarchical view of distribution of costs
  - example: quantify initialization, solve, post-processing
- Caller's view - bottom-up (up the call chain)
  - apportion a procedure's metrics to its dynamic calling contexts
  - understand costs of a procedure called in many places
  - example: see where MPI_Wait() is called from
- Flat view - ignores the calling context of each sample
  - aggregate all metrics for a procedure, from any context
  - attribute costs to loop nests and lines within a procedure
  - example: assess the overall memory hierarchy performance within a critical procedure

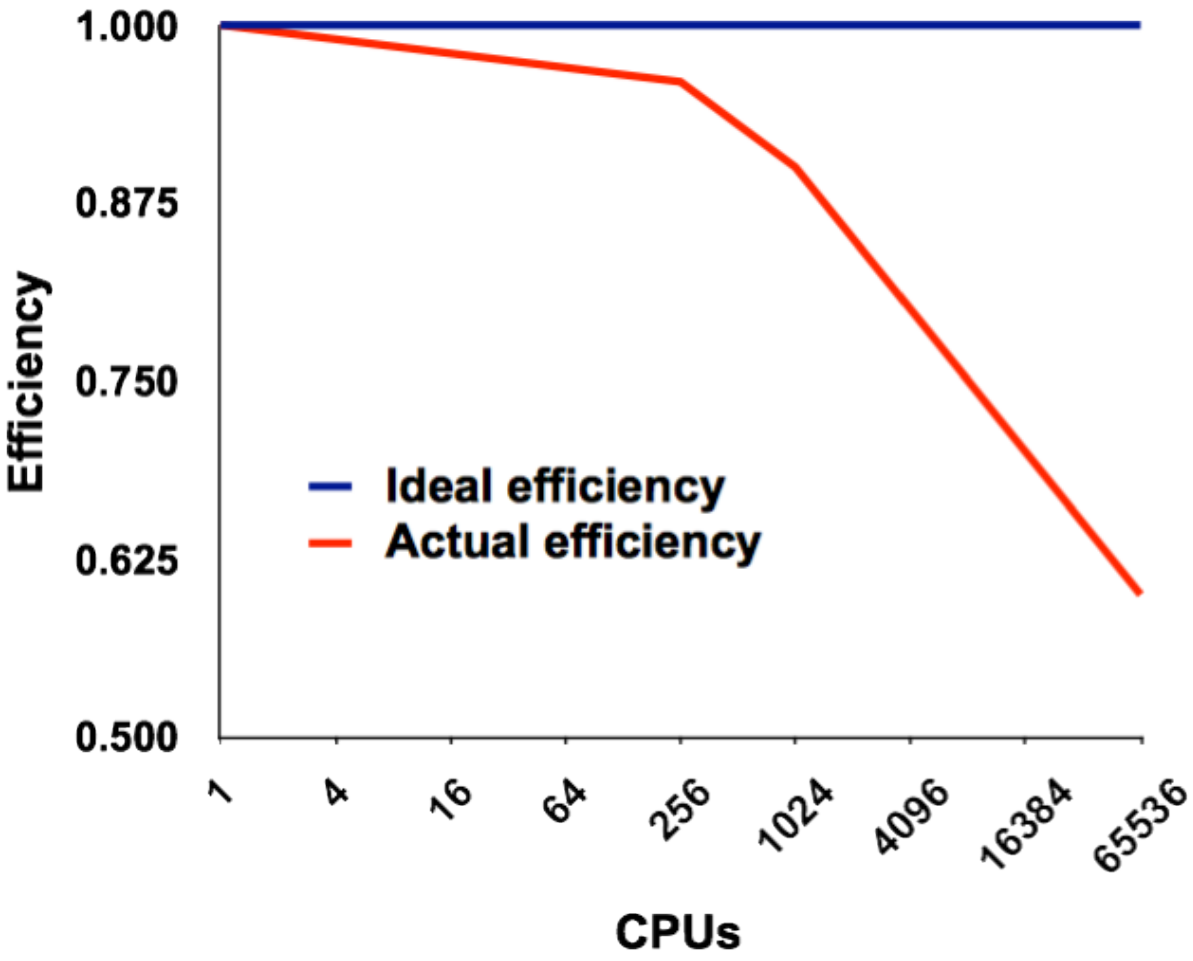

Creating a derived metric
A derived metric is a spreadsheet-like formula using other metrics (variables), operators, functions, and numerical constants.
Derived metric definition
Name:
IPC (I)
Provide a name
Formula:
$2/$0
... and a formula
There are two kinds of metric variables: point-wise and aggregate. The former is like a spreadsheet cell, the latter is like a spreadsheet-column sum. To form a variable, prepend '$' and '@', respectively, to a metric id.
For instance, the formula
(($2 - $1) * 100.0) / @1
divides the scaled difference of the point-wise metrics 2 and 1 by the aggregate value of metric 1.
Assistance:
Metrics:
0: PAPI_TOT_CYC.[0,0] (I)
Point-wise
Aggregate
Functions:
stdev(x1, x2, ..., xn)
Insert function
Operators:
( ) + - * / ^
Advanced options
Augment metric value display with a percentage relative to column total
Default format
Display metric value as percent
Custom format
The format is based on java.util.Formatter class which is almost equivalent to C's printf format.
Example: '%6.2f ' will display 6 digit floating-points with 2 digit precision.
Cancel
OK

Compute derived metrics


hpcviewer: matmul
source.c
49
50 sum = 0.0;
51 for (i=0 ; i<reps ; ++i)
52 {
53 compute();
54 sum += MAT_C(3,3);
55 if ((i % 2) == 0)
56 {
57 compute();
58 sum += MAT_C(3,3);
59 }
60 }
61
62 printf("Size, %d, %ld, %ld, %.1lf, sum=%lf\n", N, reps, reps*N*N*N, MAT_C(3,3), sum);
Calling Context View
Callers View
Flat View
Scope
IPC (I)
PAPI_TOT_CYC.[0,0] (I)
PAPI_TOT_CYC.[0,0] (E)
PAPI_TOT_INS.[0,0] (I)
PAPI_TOT_INS.[
Experiment Aggregate Metrics 1.60e+00 9.96e+09 100 % 9.96e+09 100 % 1.59e+10 100 % 1.59e+1
compute 1.60e+00 9.95e+09 99.9% 9.95e+09 99.9% 1.59e+10 100 % 1.59e+1
53: main 1.60e+00 6.64e+09 66.6% 6.64e+09 66.6% 1.06e+10 66.7% 1.06e+1
57: main 1.60e+00 3.32e+09 33.3% 3.32e+09 33.3% 5.31e+09 33.3% 5.31e+0
do_lookup_x 6.00e+06 0.1% 6.00e+06 0.1%
_dl_fixup 6.00e+06 0.1%
_dl_lookup_symbol_x 6.00e+06 0.1%
_dl_runtime_resolve 6.00e+06 0.1%
main 1.60e+00 9.96e+09 100 % 1.59e+10 100 %

Pinpointing Scalability Bottlenecks

The Problem of Scaling



Note: higher is better

Performance Analysis with Expectations

- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time
- Put your expectations to work
  - measure performance under different conditions
    - e.g. different levels of parallelism or different inputs
  - express your expectations as an equation
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively

Weak Scaling Analysis for SPMD Codes

Performance expectation for weak scaling

- work increases linearly with # processors
- execution time stays constant

$$C(n_q) = C(n_p)$$

$$X_w(n_q) = \frac{C(n_q) - C(n_p)}{T_q}$$

parallel overhead

total time

Strong Scaling Analysis for SPMD Codes

Performance expectation for strong scaling

- work is constant
- execution time decreases linearly with # processors

- Execute code on p and q processors; without loss of generality, p < q
- Let T_i = total execution time on i processors
- For corresponding nodes n_q and n_p
  - let $C(n_q)$ and $C(n_p)$ be the costs of nodes n_q and n_p
- Expectation: $$qC_q(n_q) = pC_p(n_p)$$
- Fraction of excess work: $$X_s(C,n_q) = \frac{qC_q(n_q) - pC_p(n_p)}{qT_q}$$ parallel overhead / total time

Hands-On Demo

- Using the CG benchmark from the OpenMP version of the NAS Parallel Benchmark suite 3.3.1
- Collect time measurements (`PAPI_TOT_CYC`) for different process or thread counts, e.g. 2 and 4 threads
- Produce performance database using data for thread 0 from both runs

```
$ hpcprof -S cg.hpcstruct -o db_cgB -M thread hpc_cgB_[24]/
cg.B.x-000000-000-*.hpcrun
```

- Perform differential analysis
  - Compute "`scalability loss`" metric for both inclusive and exclusive metrics

hpcviewer: cg.B.x

cg.f

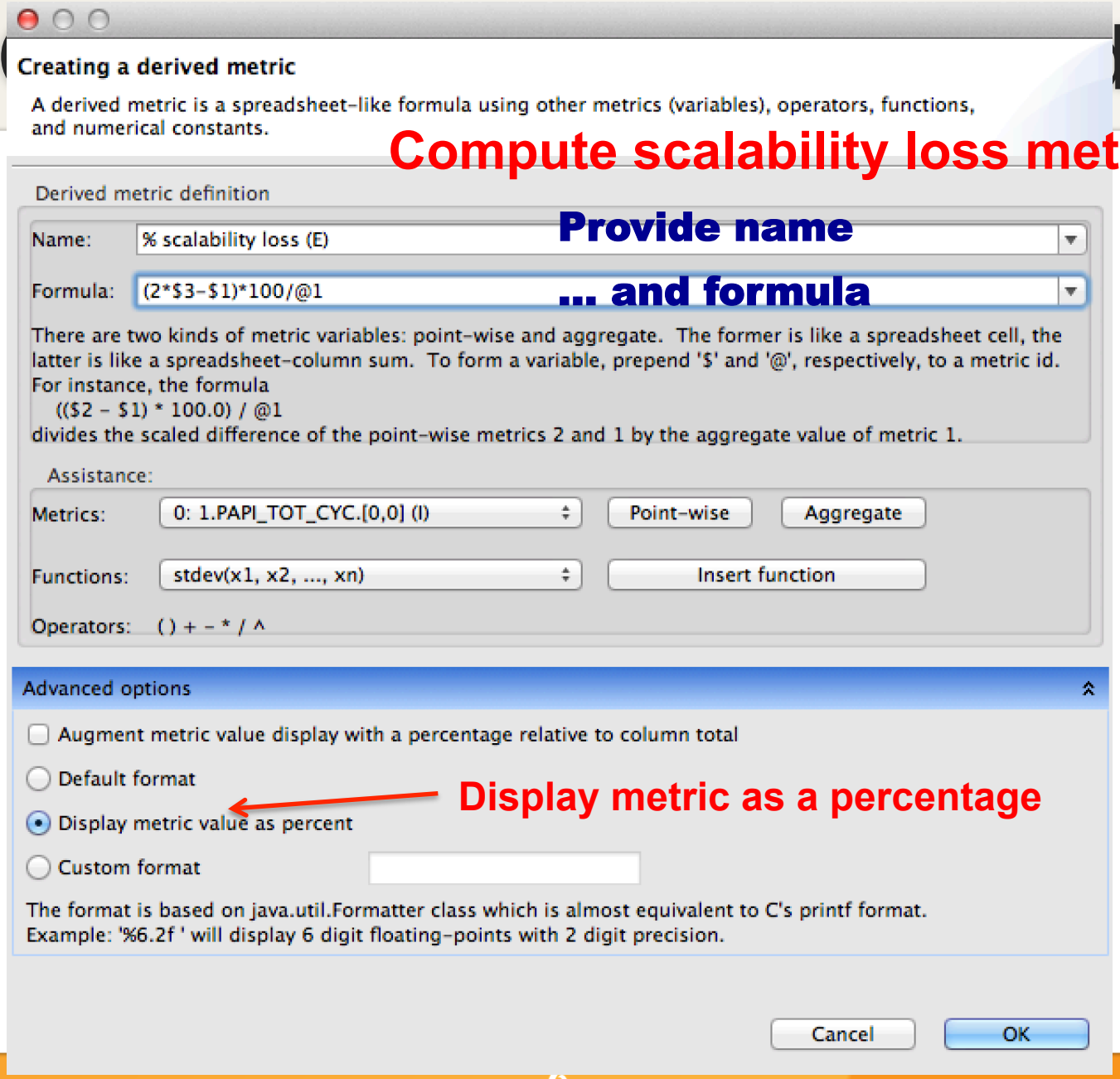
Strong scaling data

```
465
466
467 c---------------------------------------------------------------------
468 c---------------------------------------------------------------------
469       subroutine conj_grad ( colidx,
470      >                       rowstr,
471      >                       x,
472      >                       z,
473      >                       a,
474      >                       p,
475      >                       q,
476      >                       r,
477      >                       rnorm )
478 c---------------------------------------------------------------------
```

Calling Context View | Callers View | Flat View

2-thread run **4-thread run**

Scope	1.PAPI_TOT_CYC.[0,0] (I)	1.PAPI_TOT_CYC.[0,0] (E	2.PAPI_TOT_CYC.[0,0] (I)	2.PAPI_TOT_CYC.[0,0] (E)
Experiment Aggregate Metrics	1.93e+11 100 %	1.93e+11 100 %	1.24e+11 100 %	1.24e+11 100 %
conj_grad	1.87e+11 97.0%	1.74e+11 90.2%	1.21e+11 97.3%	7.72e+10 62.3%
L_kmp_invoke_pass_parms	1.87e+11 97.0%	1.74e+11 90.2%	1.20e+11 97.0%	7.72e+10 62.3%
245: cg	2.29e+09 1.2%		1.54e+09 1.2%	
319: cg	1.84e+11 95.8%		1.19e+11 96.1%	
__kmp_wait_sleep	1.33e+10 6.9%	7.91e+09 4.1%	4.39e+10 35.4%	2.58e+10 20.8%
sparse	5.49e+09 2.9%	5.38e+09 2.8%	2.83e+09 2.3%	2.71e+09 2.2%
__kmp_x86_pause	2.93e+09 1.5%	2.93e+09 1.5%	9.99e+09 8.1%	9.99e+09 8.1%
__kmp_execute_tasks	2.15e+09 1.1%	2.15e+09 1.1%	6.85e+09 5.5%	6.85e+09 5.5%
__kmp_yield	3.22e+08 0.2%	2.98e+08 0.2%	1.28e+09 1.0%	1.20e+09 1.0%
cg	1.93e+11 100 %	3.15e+07 0.0%	1.24e+11 100 %	1.40e+07 0.0%
makea	5.61e+09 2.9%	3.15e+07 0.0%	2.97e+09 2.4%	1.05e+07 0.0%


Creating a derived metric
A derived metric is a spreadsheet-like formula using other metrics (variables), operators, functions, and numerical constants.
Compute scalability loss metric
Derived metric definition
Name:
% scalability loss (E)
Provide name
Formula:
(2*$3-$1)*100/@1
... and formula
There are two kinds of metric variables: point-wise and aggregate. The former is like a spreadsheet cell, the latter is like a spreadsheet-column sum. To form a variable, prepend '$' and '@', respectively, to a metric id. For instance, the formula
(($2 - $1) * 100.0) / @1
divides the scaled difference of the point-wise metrics 2 and 1 by the aggregate value of metric 1.
Assistance:
Metrics:
0: 1.PAPI_TOT_CYC.[0,0] (I)
Point-wise
Aggregate
Functions:
stdev(x1, x2, ..., xn)
Insert function
Operators:
( ) + - * / ^
Advanced options
Augment metric value display with a percentage relative to column total
Default format
Display metric value as percent
Display metric as a percentage
Custom format
The format is based on java.util.Formatter class which is almost equivalent to C's printf format.
Example: '%6.2f ' will display 6 digit floating-points with 2 digit precision.
Cancel
OK

hpcviewer: cg.B.x

cg.f

```
568 !$omp do
569           do j=1,lastrow-firstrow+1
570              suml = 0.d0
571              do k=rowstr(j),rowstr(j+1)-1
572                 suml = suml + a(k)*p(colidx(k))
573              enddo
574              q(j) = suml
575           enddo
576 !$omp end do
577
578 CC           do j=1,lastrow-firstrow+1
579 CC              i = rowstr(j)
580 CC              iresidue = mod( rowstr(j+1)-i, 2 )
581 CC              sum1 = 0.d0
```

Calling Context View | Callers View | Flat View

Sort by loss (E)

Scope	% scalability loss (E)	1.PAPI_TOT_CYC.[0,0] (I)	1.PAPI_TOT_CYC.[0,0] (E)	2.PAPI_TOT_CYC.[0,0] (I)	2.PAPI_
Experiment Aggregate Metrics	28.65 %	1.93e+11 100 %	1.93e+11 100 %	1.24e+11 100 %	
▼ __kmp_wait_sleep	22.67 %	1.33e+10 6.9%	7.91e+09 4.1%	4.39e+10 35.4%	
▼ __kmp_barrier	22.29 %	1.31e+10 6.8%	7.79e+09 4.0%	4.32e+10 34.9%	
▼ __kmpc_barrier	15.00 %	1.24e+10 6.4%	7.40e+09 3.8%	3.08e+10 24.8%	
▶ 576: conj_grad	12.26 %	1.10e+10 5.7%	6.58e+09 3.4%	2.57e+10 20.7%	
▶ 660: conj_grad	1.28 %	3.54e+08 0.2%	1.86e+08 0.1%	2.20e+09 1.8%	
▶ 554: conj_grad	0.89 %	4.90e+07 0.0%	2.45e+07 0.0%	1.45e+09 1.2%	
▶ 679: conj_grad	0.41 %	7.66e+08 0.4%	5.00e+08 0.3%	1.12e+09 0.9%	
▶ 805: makea	0.05 %	5.95e+07 0.0%	3.85e+07 0.0%	1.12e+08 0.1%	
▶ 526: conj_grad	0.05 %			8.05e+07 0.1%	
▶ 1024: sparse	0.04 %	7.00e+06 0.0%	3.50e+06 0.0%	7.35e+07 0.1%	
▶ 896: sparse	0.02 %			2.45e+07 0.0%	

Summary

- Performance tools help us evaluate application performance
- HPCToolkit: low overhead, full-code profiler
  - Uses hardware counter sampling on top of PAPI
  - Maps performance data to functions, loops, calling contexts
  - Intuitive viewer
    - Enables top-down analysis
    - Custom derived metrics enable quick performance analysis at loop level
  - Differential analysis can identify scalability bottlenecks

Documentation

http://hpctoolkit.org/documentation.html

- Comprehensive user manual:

  http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf

  - Quick start guide
    - essential overview that almost fits on one page
  - Using HPCToolkit with statically linked programs
    - a guide for using hpctoolkit on BG/P and Cray XT
  - The hpcviewer user interface
  - Effective strategies for analyzing program performance with HPCToolkit
    - analyzing scalability, waste, multicore performance …
  - HPCToolkit and MPI
  - HPCToolkit Troubleshooting
    - why don't I have any source code in the viewer?
- Installation guide